VISIT…

THE EXPERT'S VOICE®

Beginning Apache Struts

From Novice to Professional

Arnold Doray

Apress®

Beginning Apache Struts

From Novice to Professional

Arnold Doray

Apress®

Beginning Apache Struts: From Novice to Professional

Copyright © 2006 by Arnold Doray

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-604-3

ISBN-10 (pbk): 1-59059-604-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Steve Anglin
Technical Reviewer: Kunal Mittal
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Julie M. Smith
Copy Edit Manager: Nicole LeClerc
Copy Editor: Liz Welch
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Susan Glinert
Proofreader: Lori Bring
Indexer: Valerie Perry
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

To my darling wife Lillian, for her love, encouragement, and constant support,
To my mother for always believing in me,
To my father for showing me by example that you can accomplish almost anything you put your mind to,
And to the One who gave up His life for us two thousand years ago.

Contents at a Glance

About the Author xix
About the Technical Reviewer xx
Acknowledgments xxi

PART 1 Basic Struts

CHAPTER 1 Introduction 3
CHAPTER 2 Servlet and JSP Review 7
CHAPTER 3 Understanding Scopes 17
CHAPTER 4 Custom Tags 23
CHAPTER 5 The MVC Design Pattern 37
CHAPTER 6 Simple Validation 53
CHAPTER 7 Processing Business Logic 67
CHAPTER 8 Basic Struts Tags 79
CHAPTER 9 Configuring Struts 93
CHAPTER 10 More Tags 107
CHAPTER 11 Uploading Files 129
CHAPTER 12 Internationalization 143
CHAPTER 13 Review Lab: Editing Contacts in LILLDEP 157

PART 2 Advanced Struts

CHAPTER 14 Tiles 161
CHAPTER 15 The Validator Framework 195
CHAPTER 16 Dynamic Forms 221
CHAPTER 17 Potpourri 239
CHAPTER 18 Review Lab: The Collection Facility 267
CHAPTER 19 Developing Plug-ins 277
CHAPTER 20 JavaServer Faces and Struts Shale 307

APPENDIX A Frameworks for the Model 359
APPENDIX B Commonly Used Classes 375
APPENDIX C Struts Tag Reference 381
APPENDIX D Answers 469
INDEX 479

Contents

About the Author xix
About the Technical Reviewer xx
Acknowledgments xxi

PART 1 Basic Struts

CHAPTER 1 Introduction 3

What Is a Web Application? 3
What Struts Can Do for You 4
About You 5
How to Read This Book 5
Useful Links 6

CHAPTER 2 Servlet and JSP Review 7

Lab 2: Installing Tomcat 7
Servlet Container Basics 10
Important Servlet Classes 12
JavaServer Pages (JSP) 13
Deconstructing Hello.jsp 14
Final Thoughts. 15
Useful Links 16
Summary 16

CHAPTER 3 Understanding Scopes 17

Lab 3: Scopes Quiz 18
Session and Request Scope Internals 20
Summary 21

CHAPTER 4 Custom Tags 23

Custom Tag Basics 23
How Custom Tags Are Processed 24
The Java Handler Classes 26
Helper Classes 28
The TLD File 29
Lab 4: A Temperature Conversion Tag 31
Step 1: Prepare the Development Environment and Scripts 32
Step 2: Write the Java Tag Handler 33
Step 3: Writing the Tag Library Descriptor file 33
Step 4: Amend web.xml 33
Step 5: Write Your JSP 34
Step 6: Deploy and Test 34
Professional Java Tools 35
Useful Links 35
Summary 35

CHAPTER 5 The MVC Design Pattern 37

The Registration Webapp 39
Requirement 1 39
Requirement 2 40
Requirement 3 41
Requirement 4 41
Requirement 5 44
Lab 5: MVC Quiz 45
Which Comes First? 46
Struts and MVC 47
Lifecycle of a Struts Request 48
Frameworks for the Model 49
Useful Links 50
Summary 51

CHAPTER 6 Simple Validation 53

Processing Simple Validations 53
Anatomy of ActionForm 54
Using ActionErrors 57

Lab 6: ContactForm for LILLDEP 61
Step 1: Prepare the Development Environment and Scripts 62
Step 2: Implement Getters and Setters for ContactForm 63
Step 3: Implement validate() 63
Step 4: Implement reset() 64
What Has Struts Done for You Today? 64
Summary 65

CHAPTER 7 Processing Business Logic 67

1,2,3 Action! 67
The Statelessness of Action 68
Subclassing Action 69
Business Logic in the Registration Webapp 70
Complex Validation 73
Data Transformation 74
Navigation 75
Lab 7: Implementing ContactAction for LILLDEP 76
Summary 77

CHAPTER 8 Basic Struts Tags 79

Page Processing Lifecycle 80
Evaluation, Replacement, and Sending 81
The View Component of the Registration Webapp 82
Declaring and Installing the HTML and Bean Libraries. 83
Displaying Static Text. 84
Forms and Form Handlers 85
Data Input Tags. 86
Displaying Errors. 87
Synopsis of HTML and Bean Tag Libraries 88
Lab 8: Contact Entry Page for LILLDEP 90
Useful Links 91
Summary 91

CHAPTER 9 Configuring Struts 93

The Structure of struts-config.xml 93
Configuring the Registration Webapp 94
Declaring Form Beans 95
Declaring Global Exceptions 96
Declaring Global Forwards 97

Declaring Form Handlers 98
Forwards 100
Controller Declaration 100
Message Resources 101
Declaring Plug-ins 101
Lab 9a: Configuring LILLDEP 102
Code Reuse 103
Lab 9b: The MNC Page 104
Summary 105

CHAPTER 10 More Tags 107

Best Practices 108
The 2 + 1 Remaining Struts Libraries 108
The Logic Tag Library 109
Iteration 109
Simple, Nested, Indexed, and Mapped Properties 111
Conditional Processing 113
Flow Control 114
The Nested Tag Library 115
JSTL and Struts 118
Expression Language (EL) 119
Using EL 120
The <c:out> Tag 121
The <c:forEach> Tag 122
The <c:if> and <c:choose>...<c:when> Tags 123
Struts EL Extensions 124
Lab 10a: The LILLDEP Full Listing Page 125
Step 1: Complete ListingAction 125
Step 2: Complete listing.jsp 125
Step 3: Amend web.xml 125
Step 4: Amend struts-config.xml 126
Lab 10b: Simplifying ContactForm 126
Step 1: Amend ContactForm 126
Step 2: Amend full.jsp and mnc.jsp 126
Lab 10c: Using JSTL 127
Step 1: Install the JSTL and Struts EL Tag Libraries 127
Step 2: Amend web.xml 127
Step 3: Amend listing.jsp 127
Useful Links 127
Summary 128

CHAPTER 11 Uploading Files 129

Uploading a Fixed Number of Files at Once 131
Uploading Any Number of Files 133
Lab 11: Importing Data into LILLDEP 140
Step 1: Complete ImportForm 141
Step 2: Complete import.jsp 141
Step 3: Complete ImportAction 141
Step 4: Amend struts-config.xml 142
Step 5: Compile, Redeploy, and Test Your Application. 142
Useful Links 142
Summary 142

CHAPTER 12 Internationalization 143

Character Encodings, Unicode, and UTF-8 143
Locales 146
Processing Input 146
Localizing Validations 147
Localizing Output 150
Processing Translated Application.properties Files 151
Selecting a Locale from the Browser 151
Switching Locales with a Link 153
Switching Locales with LocaleAction 154
Lab 12: LILLDEP for the Malaysian Market 154
Useful Links 155
Summary 155

CHAPTER 13 Review Lab: Editing Contacts in LILLDEP 157

Implementing the Edit Facility 158

PART 2 **Advanced Struts**

CHAPTER 14 Tiles 161

Installing Tiles 162
Tiles for Layout 163
Using Stylesheets with Layouts. 168
Tiles Components 169
Creating a Tiles Component 169

Example: The "Login" Tiles 173
Getting External Form Data 187
Lab 14: The Find Facility 187
Step 1: Set Up Tiles 189
Step 2: Write the Controller 189
Step 3: Put In the Tiles Action Mapping 190
Step 4: Make Changes to ContactAction 190
Step 5: Write the Tiles JSP 191
Step 6: Write the Tiles Definition 191
Step 7: Put In the Find Tile 192
Step 8: Deploy and Test 192
Summary 193

CHAPTER 15 The Validator Framework 195

Declaring the Validator Plug-in 196
Validator DTD Basics 197
Using the Validator Framework 198
Example: Validating RegistrationForm 199
Validating Nested and Indexed Properties 203
Using Constants 204
Client-Side Validations 205
The Standard Validators 205
Using validwhen 206
Using validwhen with Indexed Fields 207
Adding Custom Validations 209
Implementing validate() 209
Extending the Validator Framework 210
Implementing the Java Handler 211
Migrating Legacy Code 216
Localizing Validations 217
Lab 15: Using the Validator Framework in LILLDEP 218
Useful Links 218
Summary 219

CHAPTER 16 Dynamic Forms 221

Declaring Dynamic Forms 221
Declaring Simple Properties 223
Declaring Indexed Properties 223
Declaring Mapped Properties 224
Declaring Nested Properties 225

Accessing Dynamic Properties 225
Transferring Properties to a JavaBean 225
Dynamic Form Disadvantages 226
When to Use Dynamic Forms 227
Validating Dynamic Forms 227
The Registration Webapp with Dynamic Forms 228
See Ma, No Hands!: LazyValidatorForm (Struts 1.2.6+) 232
Disadvantages of Using LazyValidatorForm 234
The Hidden Power of BeanValidatorForm (Struts 1.2.6+) 235
Lab 16: Deleting Selected Contacts in LILLDEP 236
Step 1: Declare the SelectionForm Form Bean 236
Step 2: Amend listing.jsp 237
Step 3: Create the Action to Delete Contacts 237
Useful Links 237
Summary 238

CHAPTER 17 Potpourri 239

PropertyUtils 240
Using PropertyUtils 241
In a Nutshell... 242
DownloadAction (Struts 1.2.6+) 243
LocaleAction 245
IncludeAction and ForwardAction 247
In a Nutshell... 248
LookupDispatchAction 249
DispatchAction 254
MappingDispatchAction 255
In a Nutshell... 257
Using Global Forwards 258
Logging 258
In a Nutshell... 261
Using Wildcards 261
In a Nutshell... 262
Splitting up struts-config.xml 263
In a Nutshell... 266
Useful Links 266
Summary 266

CHAPTER 18 Review Lab: The Collection Facility 267

Lab 18a: The Main Collect Page 268
Lab 18b: The New Collection Page 269
Lab 18c: The Collection Listing Page 271
Lab 18d: Removing Selected Contacts 272
Lab 18e: Adding Selected Contacts 273
Lab 18f: Up and Down a Search 275
Summary 276

CHAPTER 19 Developing Plug-ins 277

The Task at Hand 277
Implementation Road Map 280
How Struts Processes Form Beans 281
Anatomy of a Plug-in 284
Implementing DynaFormsPlugIn 285
Reading XML with Apache's Digester 288
Implementing DynaFormsLoaderFactory 291
DefaultDynaFormsLoader 294
Lab 19: Test Driving the DynaForms Plug-in 302
Extra Credit Lab: Handling <set-property> 303
    Solution Outline 304
Useful Links 305
Summary 306

CHAPTER 20 JavaServer Faces and Struts Shale 307

JSF Overview 307
Shale Overview 308
Learning Struts a Waste of Time? 310
JavaServer Faces (JSF) 310
    Server-Side UI Components 311
    Request Processing Lifecycle 314
    Events and Event Listeners 316
    JSF Tag Libraries 320
    Value and Method Binding 320
    Navigation 321

Example: The Registration Webapp 322
Configuring JSF. 323
Message Resources 324
The User Backing Bean 325
The View 332
Where to Next? 336
Lab 20: The Struts-Faces Integration Library 337
Step 1: Preparing the Development Environment 337
Step 2: Install JSF, JSTL, and the Struts-Faces Integration Library 337
Step 3: Edit web.xml and struts-config.xml. 338
Step 4: Migrate Your Struts JSP Pages 339
Step 5: Migrate the <forward>s and Inputs 341
Step 6: Make Entry Points Forward to *.faces. 341
Step 7: Amend Actions if Necessary. 342
Step 8: Put in the Necessary <managed-bean> Declarations . . . 342
In a Nutshell 343
Struts Shale Preview 343
ViewController 344
Dialog Manager. 348
Integration with the Validator Framework 350
JNDI Integration 353
Reusable Views with Clay 354
Server-Side Ajax Support. 354
Test Framework 354
JSF vs. Shale vs. Struts 355
Useful Links 357
Summary 358

APPENDIX A Frameworks for the Model 359

Getting the Software 360
Lisptorq 360
Lab A: Test Driving Lisptorq 363
Step 1: Preparing the Development Environment 363
Step 2: Writing the Database Schema 363
Step 3: Specifying the Database Settings 364
Step 4: Generate the Java Files 364
Step 5: Writing the Test Program 365
Step 6: Initializing the Database 365
Step 7: Running the Test Program 365

Using Torque for the Registration Webapp 366
Using Hibernate for the Registration Webapp 367
In a Nutshell... 370
Autogenerating LILLDEP Model Classes 370
Useful Links 373

APPENDIX B Commonly Used Classes 375

javax.servlet.http.HttpServletRequest 375
javax.servlet.http.HttpSession 376
org.apache.struts.action.ActionMessage 376
org.apache.struts.action.ActionMessages 377
org.apache.struts.action.ActionErrors 377
org.apache.struts.action.ActionMapping 377
org.apache.struts.action.Action 377
org.apache.struts.action.ActionForm 378
org.apache.struts.upload.FormFile 379
org.apache.struts.tiles.ComponentContext 380
org.apache.struts.action.ExceptionHandler 380

APPENDIX C Struts Tag Reference 381

The HTML Tag Library 381
Common Attribute Sets 383
The Error Style Attribute Set (Struts 1.2.5+) 386
Struts-EL Tags for the HTML Tag Library 386
base 386
button 388
cancel 389
checkbox 390
errors 392
file 394
form 394
frame 396
hidden 397
html 398
image 399
img 400
javascript 402
link 404

messages 405s
multibox 407
radio 408
reset 409
rewrite 410
select, with option, options, and optionsCollection 411
submit 418
text/password 418
textarea 420
xhtml 420
The Bean Tag Library 421
Struts-EL Tags for the Bean Tag Library 422
cookie/header/parameter 422
define 423
include 425
message 426
page 429
resource 430
size 431
struts 432
write 434
The Logic Tag Library 436
Common Attribute Sets 436
Selector Attributes 437
Struts-EL Tags for the Logic Tag Library 437
empty/notEmpty 437
equal/notEqual 439
forward 440
greaterEqual/lessEqual/greaterThan/lessThan 441
iterate 442
match/notMatch 444
messagesPresent/messagesNotPresent 446
present/notPresent 447
redirect 449
The Nested Tag Library 451
Struts-EL Tags for the Nested Tag Library 452
nest 453
writeNesting 454
root 456

The Tiles Tag Library 457
Common Attributes 458
A Note on Equivalent Tags 458
Struts-EL Tags for the Tiles Tag Library 458
insert 459
definition 460
put 461
putList and add 462
get 463
getAsString 464
useAttribute 465
importAttribute 466
initComponentDefinitions 467

APPENDIX D Answers 469

Chapter 1: Introduction 469
Chapter 3: Understanding Scopes 470
Chapter 5: The MVC Design Pattern 471
Chapter 6: Simple Validation 471
Lab 8: Contact Entry Page for LILLDEP 472
Lab 9a: Configuring LILLDEP 473
Lab 9b: The MNC Page 473
Chapter 10: More Tags 474
Lab 10a: The LILLDEP Full Listing Page 475
Lab 10b: Simplifying ContactForm 475
Lab 11: Importing Data into LILLDEP 475
Chapter 13: Review Lab 476
Chapter 14: Tiles 476
Chapter 15: The Validator Framework 477
Chapter 17: Potpourri 477
Chapter 20: JavaServer Faces and Struts Shale 477

INDEX 479

About the Author

ARNOLD DORAY is the lead software architect of Thinksquared, an IT consultancy based in Singapore. He has been developing software professionally for nearly a decade, and has conducted developer training courses in the UK and Singapore. This book has its roots in a Struts course he teaches. Besides coding, Arnold loves physics, long-distance cycling, and trekking in the jungles of Malaysia.

About the Technical Reviewer

KUNAL MITTAL is a consultant specializing in Java technology, the J2EE platform, web services, and SOA technologies. He has coauthored and contributed to several books on these topics. Kunal is a director in the information technology group within the Domestic TV division of Sony Pictures Entertainment. In his spare time he does consulting gigs for startups in the SOA space and for large companies looking to implement an SOA initiative. You can contact Kunal through his website at `www.soaconsultant.com` or at `kunal@kunalmittal.com`.

Acknowledgments

Any book is the product of a team effort, and this book is certainly no exception. I'm especially grateful to Kunal Mittal, who besides technically reviewing this book, made many suggestions that significantly improved its readability and utility. I'd like to thank Julie Smith for being the perfect project manager—giving support when needed and making sure that everything ran on schedule with a minimum amount of fuss—and all this through email!

Liz Welch did a great job of adding professional polish to the text and I'd like to thank her for being a *huge* help during the copyediting stage. Of course, any remaining errors are entirely the fault of my spellchecker! I'd like to thank Katie Stence for her patience in accommodating my many final changes.

My sincere thanks to Steve Anglin, who also saw the need for this book and agreed to let me write it.

Finally, this book would not be possible without the support of my wife Lillian, who endured many months with an absent husband. My humble thanks to you, my dearest.

PART 1

Basic Struts

Struts grew out of a personal need (open source developers often call this scratching your own itch*) to support the development of an application that I was responsible for... I began the process of open sourcing this idea of a web application framework at the Apache Software Foundation. What happened next was nothing short of extraordinary—Struts quickly became the de facto standard web application architecture in the J2EE space (the number of significant Internet applications built with it is substantial, but is dwarfed by the number of intranet applications that use it), integrated into nearly all the major app servers and tools, supported by a rich ecosystem of knowledgeable professionals and skilled developers, backed by significant documentation in the form of books and articles, and the basis for a large user community centered around the Struts User mailing list...*

—Craig McClanahan

CHAPTER 1

Introduction

This book describes a web application framework called Apache Struts, with which you can *easily* build *robust*, *maintainable* web applications. Now, the three italicized adjectives aren't just hype:

- **Simplicity**: Struts is relatively easy to pick up and use effectively. You get a lot out of Struts for the effort you put in to learn it.
- **Robustness**: Struts provides much of the infrastructure you need to create webapps. You can rely on this tested body of code built on solid design principles, instead of having to come up with solutions of your own.
- **Maintainability**: Struts is built around the Model-View-Controller (MVC) design pattern, which promotes separation between various portions of your webapp. This goes a long way to making your code maintainable.

Besides this, Struts has a huge user base and is a vital component in many Java-based enterprise web application solutions. It's here to stay for some time to come.

Just to make sure we're all on the same page, it might be helpful to understand what a web application is.

What Is a Web Application?

In this book, the term *web application*, or *webapp*, refers to any application whose user interface is hosted on a web browser. It is useful to think of webapps as falling somewhere on a continuum (see Figure 1-1). At one end of this continuum are webapps that deliver static content. Most websites are an example of this. On the other extreme are webapps that behave just like ordinary desktop applications. Struts is useful in building webapps on this right half of the continuum.

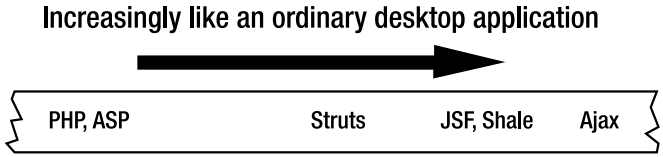

Figure 1-1. *The continuum of web application frameworks*

So, if you're planning to build a webapp requiring little or no user input or input validation, no data access, no complex business rules, or no need for internationalization, you probably don't have to use Struts. If some or all of these are necessary, then your application would definitely benefit from using Struts.

■ **Note** I hope Figure 1-1 isn't cause for confusion. There are many dimensions on which to place a particular framework (ease of use, support for a specific feature, cost, platform independence, etc.), and Figure 1-1 places the technologies on just one dimension.

I should warn you that the *extreme* right half of the continuum is not something Struts fully addresses either. Emerging technologies like JavaServer Faces (JSF) and Ajax (Asynchronous JavaScript and XML) attempt to address this deficiency.

Similarly, some web applications might require a mix of technologies to work, with Struts as a basic ingredient. The use of Enterprise JavaBeans (EJBs) for processing of business logic, or a template engine like Velocity to process text, are examples. In most cases, though, plain ol' Struts will do the job.

In the course of this book, I'll give you an overview of a number of technologies that work with Struts, among them Torque, Hibernate (in Appendix A), and JSF (which I'll cover in detail in Chapter 18, along with Shale).

What Struts Can Do for You

Craig McClanahan created Struts because he was asked to port a U.S.-centric application to Europe. He had to give the application a web interface as well as allow it to be easily translated to four European languages (for this and other interesting bits, see Craig's blog in "Useful Links" at the end of this chapter). So, needless to say, internationalization is something that Struts makes particularly easy to do. But Struts has much to offer beyond this.

Struts provides an extensive infrastructure for things like user input validation, error handling and reporting, and flow control, all of which are the bread and butter of building webapps. In my experience, about 30–50 percent of the time (and 100 percent of the tedium)

used to build webapps revolves around these activities, so trimming it down substantially is a Very Good Thing Indeed.

Struts allows people with different skill sets—web designers, system analysts, system engineers, database engineers—to work in parallel. Struts does this by enforcing a "separation of concerns": pieces of code that do different things are bundled separately and given standardized ways of communicating between themselves. This way, people can maintain or work on some parts of the webapp without having to worry too much about other parts. Can you see why?

Even if you're a lone developer, Struts' enforcement of a separation of concerns translates into modularized and therefore more maintainable code. Although this might sound like something you've always been doing in Java, Struts helps you do it across a smorgasbord of languages—Java, JSP, JavaScript, and SQL.

Struts does all this *and* is easy to learn, which certainly contributes to its popularity. There are, in fact, other web application frameworks, both open source and proprietary, but Struts has seen explosive growth over competing frameworks and now has a very large user base. So, if you're ever stuck with a sticky Struts problem, you have ready access to a very large body of knowledge on the Internet.

About You

In writing this book, I've been presumptuous enough to make a few assumptions about your programming background. In particular, I've assumed that you have

- **No experience with servlets**: If you *have* worked with servlets, then you have an advantage, but not much, since Struts does an admirable job of hiding most of the underlying servlet infrastructure.

- **Some experience with JSP**: If you've only used ASP or PHP, not to worry; you should be able to pick up JSP essentials as the book progresses.

- **A working knowledge of Java**: You don't have to be a Java expert, but a sound knowledge of Java (1.4) basics is absolutely essential. If you can't tell when to use a `Reader` and when to use an `InputStream`, you might find this book difficult to follow.

How to Read This Book

A friend of mine once compared programming to carpentry. I couldn't agree more. Perhaps the only way to acquire new skills in either discipline is by application to nontrivial projects. Would you buy furniture built by someone who "learned" carpentry through correspondence?

In the same spirit, this book takes a very hands-on approach in teaching you Struts. The lab sessions and exercises are an *integral* part of the book.

■ **Note** I can't overemphasize this: You won't learn a thing from this book if you don't attempt the labs!

Having said that, I've tried to make the lab sessions challenging and interesting, but not overly complicated. Past the first few preliminary chapters, subsequent lab sessions involve building a nontrivial, real-life data entry and display application. I hope you enjoy seeing the final product emerge as you progress.

Every lab session has solutions, which have been carefully checked and independently reviewed, and which can be found in the Source Code section of the Apress website (`http://www.apress.com`). Questions raised in either the lab sessions or in the main text are in Appendix D.

There are two parts to this book, and each part ends with "review" labs, to help you solidify what you've learned.

The first part of the book covers Struts basics, as well as JSP and Servlet prerequisites. The second part of the book covers the more "advanced" portions of Struts like Tiles, the Validator framework, and how to build your own plug-ins, among other things. We'll also study JSF and Struts Shale in the latter half of the book.

Useful Links

- You'll find the Apache Struts website at `http://struts.apache.org`.
- Ted Husted's website, `http://husted.com/struts`, contains many nuggets on Struts best practices.
- The ServerSide, `www.theserverside.com`: has useful articles and discussions on Struts.
- You'll find Craig McClanahan's blog at `http://blogs.sun.com/roller/page/craigmcc`.
- Spring is another web application framework; learn about it at `www.springframework.org`.
- *Pro Jakarta Struts*, from Apress, is now in its second edition: `http://www.apress.com/book/bookDisplay.html?bID=228`.
- Also check out *Foundations of Ajax* (`http://www.apress.com/book/bookDisplay.html?bID=10042`) and P*ro Jakarta Velocity:From Professional to Expert* (`http://www.apress.com/book/bookDisplay.html?bID=347`), both from Apress.

CHAPTER 2

Servlet and JSP Review

Struts is built on top of servlet technology, which is a basic infrastructure for building webapps with the Java platform.

This technology is really a specification laid out by Sun, detailing the behavior of a program called a **servlet container**. A servlet container takes care of parsing HTTP requests, managing sessions, and compiling JavaServer Pages (JSPs), among other things.

Servlets are Java classes you write in order to handle HTTP requests and responses. These classes are usually subclasses of `HttpServlet`, which has many convenience functions to help your servlet subclasses do their job.

Anyone can implement Sun's Servlet specification, and in fact, there are many servlet containers available. Popular ones include WebLogic from IBM and the freely available, open source Tomcat from the Apache Software Foundation (ASF).

One of the beauties of Struts is that although it is dependent on servlet technology, you can develop Struts applications with having to know much about servlets! This chapter will equip you with everything you need to know about servlets for the journey ahead.

The first thing you need to do is install a servlet container on your machine.

Lab 2: Installing Tomcat

In this lab session, you'll install the Tomcat servlet container on your PC. Be sure to finish this lab because you will be using Tomcat in subsequent chapters. Complete the following:

1. Choose a convenient location on your hard drive on which to perform the installation. Let's call this the `INSTALL` directory. For example, you might want to use `C:\Tomcat\` as an `INSTALL` directory.

2. From the Source Code section of the Apress website, found at `http://www.apress.com`, copy the file `jakarta-tomcat-5.0.28.zip` to the `INSTALL` directory.

■ **Note** If you're already using Tomcat, be sure you're using 4.*x* or (better) 5.0.*x*, because higher versions (5.5.*x* and above) require JSDK 1.5 and versions lower than 4.*x* require some fiddling to work with Struts.

---

3. Unzip the contents of the zip file, ensuring that the Create Folders option is selected. The directory `jakarta-tomcat-5.0.28` should appear.

4. Locate the directory of your Java SDK installation. This directory is known as the `JAVA_HOME` directory.

■ **Note** I'm assuming you have installed the 1.4 JSDK.

---

5. Open `.\jakarta-tomcat-5.0.28\bin\startup.bat` with a text editor such as Notepad and put in an extra line at the start to set up the `JAVA_HOME` path. For example, if the JDK is in `C:\jdk1.4.2\`, you'd use

```
JAVA_HOME=C:\jdk1.4.2\
```

6. Repeat Step 5 for the `shutdown.bat` batch file.

■ **Note** If you're using Linux or some *nix OS, the files to amend are `startup.sh` and `shutdown.sh`.

Now test your Tomcat installation:

7. Double-click on the `startup.bat` batch file. You should see a DOS screen appear (see Figure 2-1).

8. Open the URL `http://localhost:8080` with your favorite web browser. You should see the default Tomcat welcome web page (Figure 2-2).

9. When you're done, double-click the `shutdown.bat` batch file to stop Tomcat.

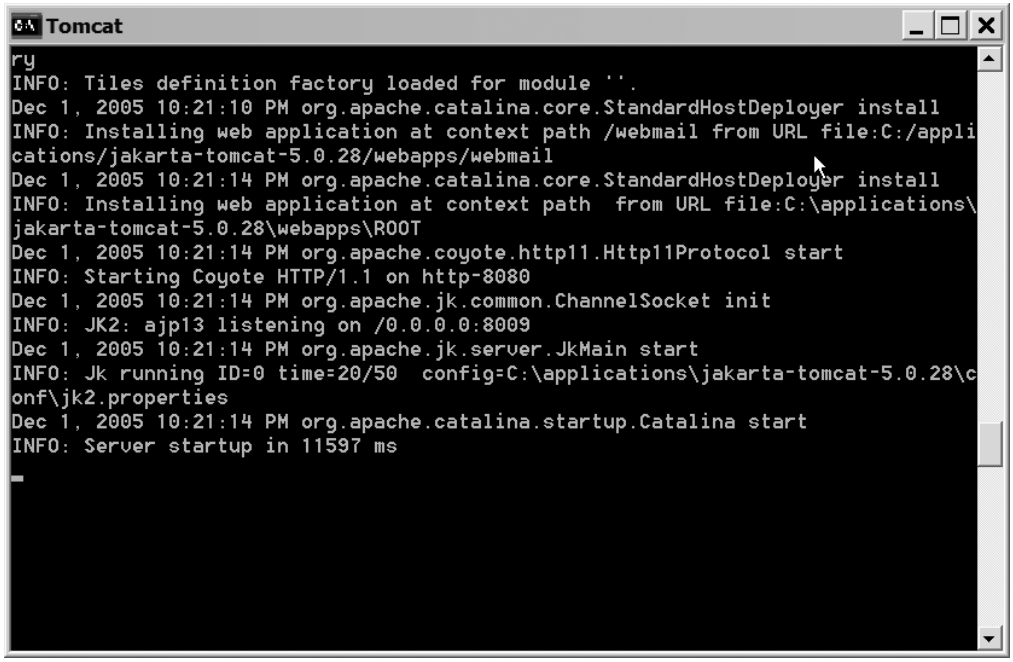
Tomcat
ry
INFO: Tiles definition factory loaded for module ''.
Dec 1, 2005 10:21:10 PM org.apache.catalina.core.StandardHostDeployer install
INFO: Installing web application at context path /webmail from URL file:C:/appli
cations/jakarta-tomcat-5.0.28/webapps/webmail
Dec 1, 2005 10:21:14 PM org.apache.catalina.core.StandardHostDeployer install
INFO: Installing web application at context path from URL file:C:\applications\
jakarta-tomcat-5.0.28\webapps\ROOT
Dec 1, 2005 10:21:14 PM org.apache.coyote.http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on http-8080
Dec 1, 2005 10:21:14 PM org.apache.jk.common.ChannelSocket init
INFO: JK2: ajp13 listening on /0.0.0.0:8009
Dec 1, 2005 10:21:14 PM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=20/50 config=C:\applications\jakarta-tomcat-5.0.28\c
onf\jk2.properties
Dec 1, 2005 10:21:14 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 11597 ms

Figure 2-1. *Tomcat starting up*

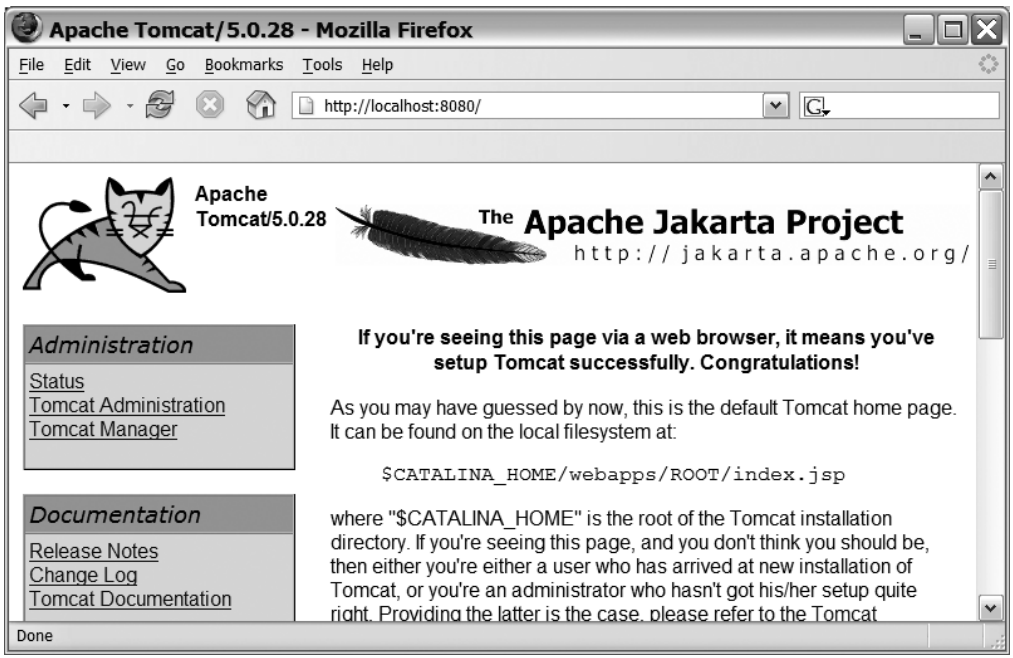
Apache Tomcat/5.0.28 - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/
Apache
Tomcat/5.0.28
The Apache Jakarta Project
http://jakarta.apache.org/
Administration
Status
Tomcat Administration
Tomcat Manager
Documentation
Release Notes
Change Log
Tomcat Documentation
If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!
As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:
$CATALINA_HOME/webapps/ROOT/index.jsp
where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then either you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite
Done

Figure 2-2. *The Apache Tomcat welcome page*

SHOULD YOU ENCOUNTER PROBLEMS...

If you encounter problems, check the following:

- That you've got the `JAVA_HOME` path right. If `startup.bat` terminates abruptly, this is likely the problem.
- That you don't have another web server (like IIS or PWS) or application latched on to port 8080.
- If you have a firewall running on your PC, ensure that it isn't blocking traffic to and from 8080. Also ensure that it doesn't block `shutdown.bat`.

If none of this works, try putting a `pause` DOS command at the end of the batch file to read the error message. Try using Google with the error message. Nine times out of ten, someone else has had the same problem and posted a solution on the Internet.

Servlet Container Basics

Now that you have installed Tomcat, you have to know how to install your web applications on it. The information in this section applies to any conformant servlet container, not just Tomcat.

Apart from the `bin` directory in which `startup.bat` and `shutdown.bat` live, the most other important directory is the `webapps` directory, which will contain your web applications.

A web application is bundled as a WAR file (for **W**eb **A**pplication a**R**chive), which is just an ordinary JAR file with the `.war` extension. To deploy a web application, you simply drop its WAR file in the `webapps` directory and presto! Tomcat automatically installs it for you. You know this because Tomcat will create a *subdirectory* of `webapps` with the same name as your web application.

■ **Note** If you want to **re-deploy** a web application, you have to **delete** the existing web application's subdirectory. Most servlet containers will not overwrite an existing subdirectory.

The WAR archive file must contain the following:

- A `WEB-INF` directory, into which you put a file called `web.xml`, used to configure your webapp. Struts comes with a `web.xml` file that you must use.
- A `WEB-INF\lib` directory, into which you put all Struts JAR files, including any third-party JAR files your application uses.

- A `WEB-INF\classes` directory, into which go the `.class` files for your webapp.
- Your JSP, HTML, image, CSS, and other files. JSP and HTML files are usually on the "root" of the WAR archive, CSS usually in a `styles` subdirectory, and images in a (surprise!) `images` subdirectory.

Figure 2-3 illustrates this information.

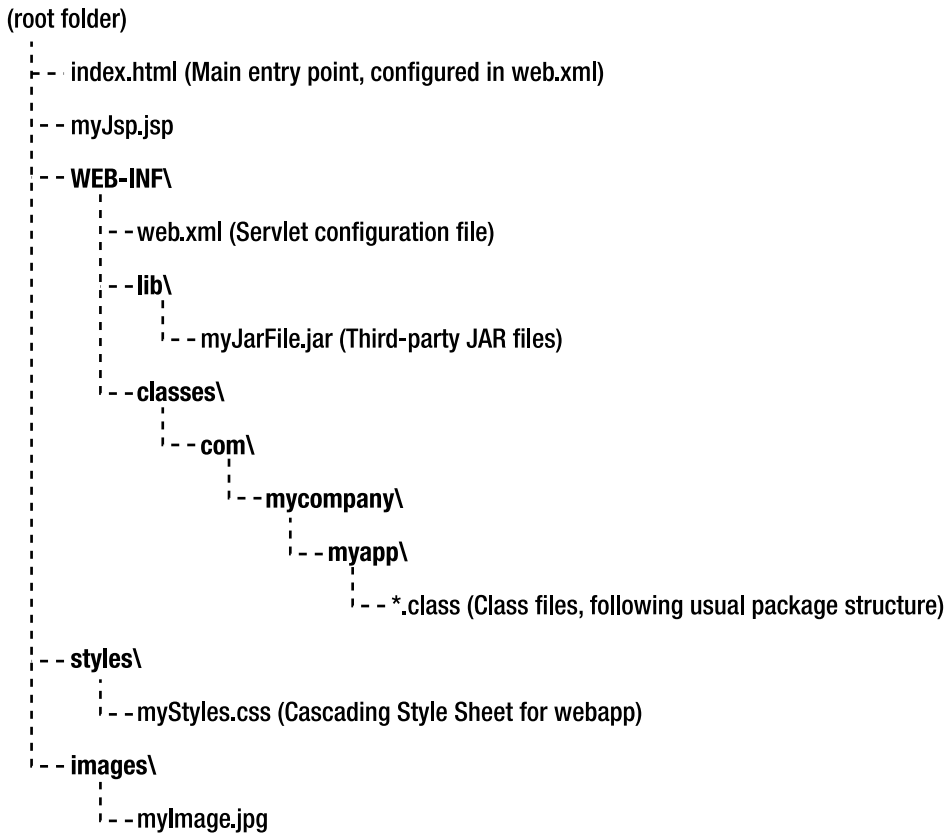

Figure 2-3. *Structure of a simple WAR file*

Don't worry too much about having to memorize these details for the time being. Future labs should make them second nature to you.

Important Servlet Classes

In the introduction to this chapter, I told you that you didn't have to know much about servlets in order to use Struts.

This is true, except for two servlet helper classes that are very important to Struts. You need to pay special attention to these. They will certainly be used in your Struts applications:

- `HttpServletRequest`: Used to read parameter values on an incoming HTTP request and read or set "attributes" that you can access from your JSPs.

- `HttpSession`: Represents the current user session. Like `HttpServletRequest`, `HttpSession` also has get and set functions for attributes but they are associated with the user session instead of the request.

These objects represent "scopes," or lifetimes, of variables on your webapp. I'll explain scopes in more detail in the next chapter.

Figure 2-4 illustrates the most used functions on these two classes.

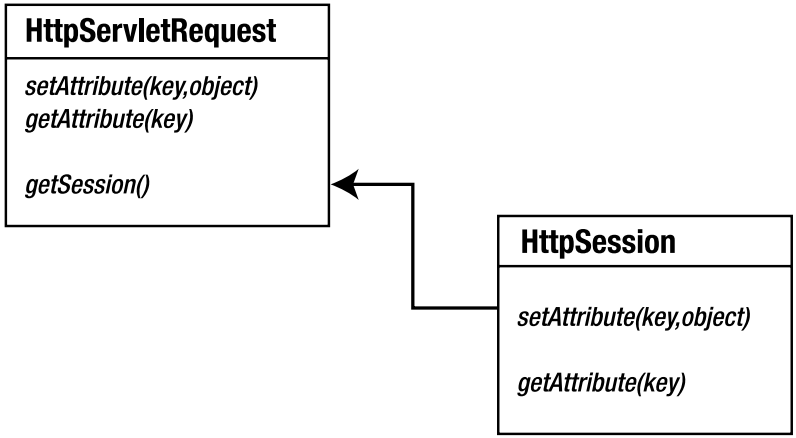

Figure 2-4. *HttpServletRequest and HttpSession*

Note that you can obtain an instance of `HttpSession` from `HttpRequestServlet` using the latter's `getSession()` function. The important functions on both classes are also listed in Appendix B.

JavaServer Pages (JSP)

Earlier, I mentioned that servlets are Java classes that generate an appropriate response to an HTTP request. This response is usually an HTML page displaying the results of a computation.

Unfortunately, a pure Java class is often not the easiest way to generate HTML. Your typical web page contains lots of text, and your servlet subclass would have to contain or access this text somewhere. In the worst-case scenario, you'd have Java classes containing lots of text and HTML tags. And don't forget, you'd have to escape every double quote. Trust me, it's not a pretty sight.

The problem is that Java classes (and therefore servlet subclasses) are *code-centric,* and are not geared to allow the programmer to easily display large quantities of text, like a web page.

JavaServer Pages (JSP) is a solution to this problem. A JSP page is *text-centric*, but allows developers full access to the Java programming language via *scriptlets*. These consist of Java code embedded in the JSP file between special marker tags, `<% ... %>`, as shown in Listing 2-1.

■ **Note** The servlet container (like Tomcat) actually converts a JSP page to a regular servlet, then compiles it behind the scenes. This happens every time the JSP page is changed.

In the non-Java world, the closest analog to JSP would be PHP or Active Server Pages (ASP).

Listing 2-1 illustrates a simple JSP page, which I'll briefly analyze in the following subsections:

Listing 2-1. *Hello.jsp*

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/tags/time" prefix="time" %>
<html>

   <h1>Hello <%= "world".toUpperCase() %></h1>, and the time is now
   <time:now format="hh:mm"/>.

</html>
```

Figure 2-5 shows how `Hello.jsp` might appear in the web browser.

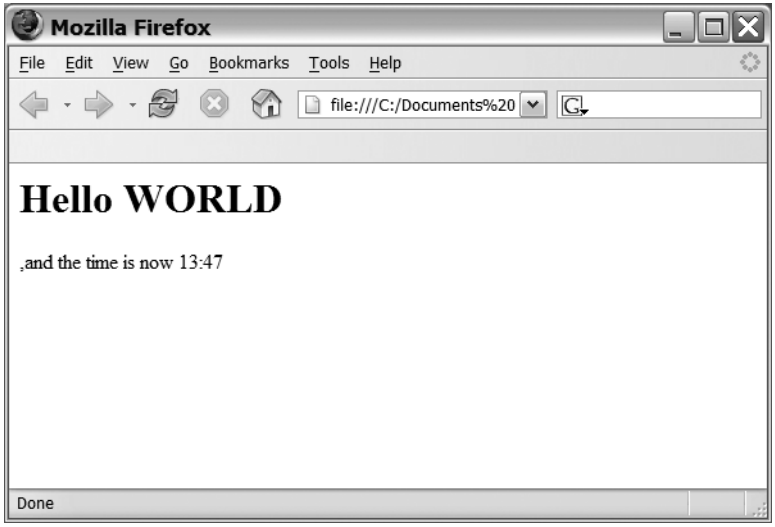

Figure 2-5. *Hello.jsp viewed in a web browser*

Deconstructing Hello.jsp

The first line in `Hello.jsp` is a header, indicating the type of output (`text/html`, using UTF-8 encoding), and the scripting language used (Java).

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
```

Immediately following this header is a tag library declaration:

```
<%@ taglib uri="/tags/time" prefix="time" %>
```

This declaration tells the servlet container that this JSP page uses a tag library. A tag library defines *custom tags* that you can create and use to put some prepackaged functionality into a JSP page.

■ **Note** In this book, I'll use the term *custom tag*, but the correct term is *custom action*. I've avoided being politically correct because "action" is an often-used word in Struts. Also, I find the term *custom tag* more intuitive.

In this instance, the prepackaged functionality is a time display:

```
<time:now format="hh:mm"/>
```

I should warn you that this "`time`" tag library is purely fictitious. I made it up just to show you how custom tags might be used. Struts uses custom tags extensively, and I'll cover this in detail in Chapter 4.

Finally, as Listing 2-1 shows, I've made gratuitous use of a scriptlet:

```
<%= "world".toUpperCase() %>
```

This is simply an embedding of Java code in the JSP page. Notice the all-important equal sign (=) right after the start of the scriptlet tag <%. This means the evaluated value of the Java code is to be displayed. It is also possible to have scriptlets that do not display anything but instead produce some side effect, as shown in Listing 2-2.

Listing 2-2. *Scriptlets in Action*

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%
  double g = (Math.sqrt(5) + 1)/2;
%>
The golden ratio is: <%= g %>
```

In Listing 2-2 there are two scriptlets. The first silently calculates a number (no = sign), and the second displays the calculated value (with an = sign).

The other instructive thing about Listing 2-2 is that it demonstrates how to define a variable in one scriptlet and access it in other scriptlets within the same page.

■ **Note** Every JSP page has a predefined variable called `request` that is the `HttpServletRequest` instance associated with the current page.

The current trend in JSP has been to try to replace the use of scriptlets with custom tags. This is largely because using scriptlets results in less readable JSPs than using custom tags. I'll revisit this idea again in Chapter 10, where I'll introduce the JSP Standard Tag Library (JSTL).

Final Thoughts

This section should be old hat to readers who have even a little experience using JSP. If you're from the ASP/PHP camps, you should have little difficulty mapping your knowledge in those technologies to JSP.

The next two chapters give you an in-depth look at two aspects of JSP technology that even many experienced JSP developers might not grasp fully, but that are crucial to your understanding of Struts.

Useful Links

- The Tomcat website `http://jakarta.apache.org/tomcat/`
- Further information on servlets: `http://java.sun.com/products/servlet/docs.html`
- The Servlet specification: `http://java.sun.com/products/servlet/download.html`
- *Pro Jakarta Struts* (Apress 2005): `http://www.apress.com/book/bookDisplay.html?bID=228`

Summary

- Tomcat's `.\webapps` directory is where you deploy your web application.
- Webapps are deployed as WAR files, which are just JAR files with a `.war` extension.
- WAR files contain a `.\WEB-INF` directory, with a `web.xml` file to configure Tomcat.
- Your Java classes and third-party JAR files go into `WEB-INF\classes` and `WEB-INF\lib`, respectively.
- Two very important servlet classes are `HttpServletRequest` and `HttpSession`.

CHAPTER 3

Understanding Scopes

In servlet-speak, a variable's "scope" refers to its *lifetime* in a web application—how long the variable exists in the course of a user's interaction with the system. A variable's scope also determines its *visibility*—who can read and change the variable. Web applications routinely utilize this concept, so you must become thoroughly familiar with it. You'll do just that in this chapter.

Servlet technology uses four types of scope. Listed in decreasing lifetimes, they are as follows:

- **Application scope**: Application-scoped variables are visible to *every* page of the web application, *all* the time. The variable is visible to *all* users of the system and exists across multiple user sessions. In other words, different users can potentially view and change the same variable at any time and from any page.

- **Session scope**: Session-scoped variables exist only for a single-user session, but they are visible to every page in that session. Users would **not** be able to view or change each other's instance of the variable.

- **Request scope**: Request-scoped variables exist only for a single request. This includes page redirects. So, if request-scoped variable `x = 7` on the first page is redirected to a second page, then `x = 7` on the second page too.

- **Page scope**: A page-scoped variable exists only on the current page. This does **not** include redirects. So, if a page-scoped variable `x = 8` on the first page is redirected to a second page, then `x` is undefined on the second page.

Note A **redirect** occurs when you ask for one page but are given another. The page you asked for redirects you to the page you actually get. You can do this in JSP with the `<jsp:forward>` tag, which you will see in action in Lab 3 later in this chapter.

Session and **request** scopes are the two most important scopes in Struts. To see if you really understand these ideas, try the following lab.

Lab 3: Scopes Quiz

Consider the following two JSP pages (Listings 3-1 and 3-2) and assume they've been loaded onto a servlet container. Your job is to predict the output on the screen as two users, Audrey and Brenda, interact with the system.

Listing 3-1. *First.jsp*

```
<jsp:useBean id="myVariable"
          scope="application" class="java.lang.StringBuffer" />
<%
   myVariable.append("1");
%>
<jsp:forward page="Second.jsp" />
```

Listing 3-2. *Second.jsp*

```
<jsp:useBean id="myVariable"
          scope="application" class="java.lang.StringBuffer" />
<%
   myVariable.append("2");
%>
<%= myVariable.toString() %>
```

The `<jsp:useBean>` tag simply defines a new JSP variable, with the given scope. `<jsp:forward>` redirects the user to a given page. Try to answer the following questions:

- Audrey is the first person to view `First.jsp`. What will she see?
- Brenda next views `First.jsp` from a different machine. What does she see?
- If Audrey again views `First.jsp` *after* Brenda, what will she see?
- What if Brenda now loads `Second.jsp` directly?

Compare your answers to the ones in Appendix D.

When you're sure you understand how application scope works, consider what happens if we change the `scope` attribute in both `First.jsp` and `Second.jsp` to `session`, then `request`, then `page`. In each instance, check your answers before proceeding with the next scope.

I hope this little *Gedankenexperiment* gives you a clear understanding of how the various scopes work. To sweep away any lingering doubts you might have, try doing it in real life:

1. Copy the `lab3.war` file found in the Source Code section of the Apress website, located at `http://www.apress.com`, into the Tomcat `\webapps\` directory.

2. Start Tomcat, if it isn't already running.

3. With your favorite web browser, navigate to `http://localhost:8080/lab3/`.

You should see a main page with links to four versions of `First.jsp`, one for each scope (see Figure 3-1). Play around with the application until you're *absolutely* sure you understand the difference between the various scopes.

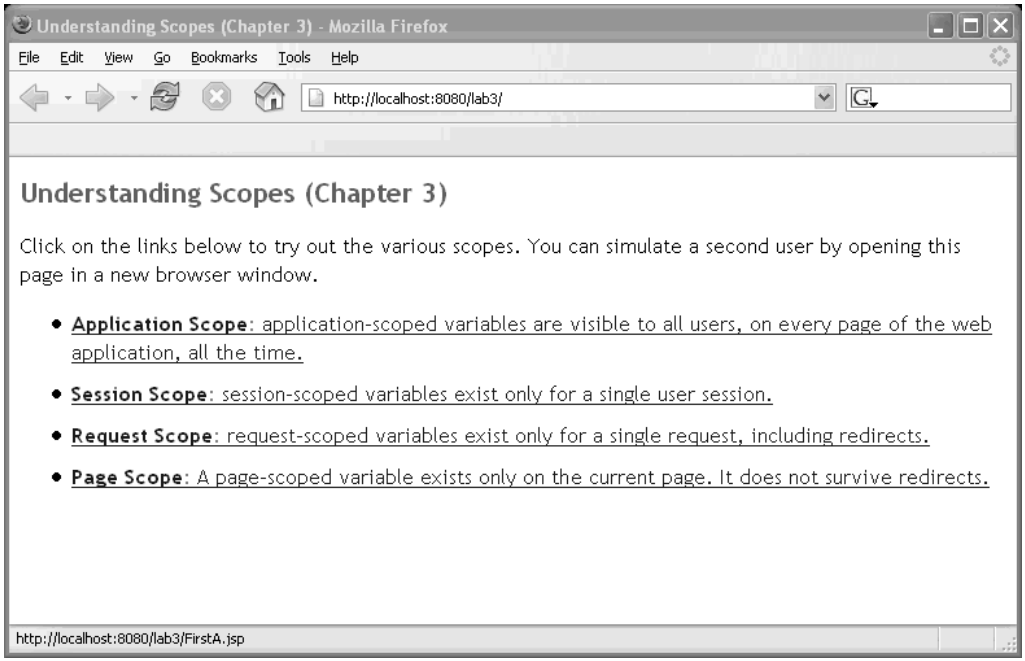
Understanding Scopes (Chapter 3) - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/lab3/
Understanding Scopes (Chapter 3)
Click on the links below to try out the various scopes. You can simulate a second user by opening this page in a new browser window.
Application Scope: application-scoped variables are visible to all users, on every page of the web application, all the time.
Session Scope: session-scoped variables exist only for a single user session.
Request Scope: request-scoped variables exist only for a single request, including redirects.
Page Scope: A page-scoped variable exists only on the current page. It does not survive redirects.
http://localhost:8080/lab3/FirstA.jsp

Figure 3-1. *The Scopes web application start page*

One last question: If we created a variable with a scriptlet on our JSP page like so:

```
<% int x = 7; %>
```

what scope would it implicitly have?

IN CASE YOU WERE WONDERING...

You might have noticed that we used a `StringBuffer` for `myVariable`, and you might be wondering if we could have used a `String` instead. We can't.

The reason is that you cannot reassign a variable declared with `<jsp:useBean>`. A `String` is immutable, and so each time you try to change a `String` with a statement like `myVariable += "1"`, you're attempting to reassign the variable.

However, calling a function on a variable is OK since we're not reassigning anything. So, `myVariable.append("1")` is acceptable, since we're just calling a function on the object referred to by the variable. The variable refers to the same object instance as it did before—no reassignment done.

Session and Request Scope Internals

In the previous chapter, we came across two servlet classes, `HttpServletRequest` and `HttpSession`, which as I explained represent the lifetimes of objects on your webapp. If this comment seemed a little cryptic at that time, I'd like to clear the air now.

Both classes have `setAttribute()` and `getAttribute()` functions (see Appendix B for details) to store and retrieve objects on them, very much like a `HashMap`'s `put()` and `get()` functions. Objects stored on `HttpServletRequest` have request scope while objects stored on `HttpSession` have session scope.

When you create a variable on your JSP page, as in Listing 3-1, you're implicitly calling `setAttribute()` on the appropriate object (an instance of either `HttpServletRequest` or `HttpSession`) depending on the scope you used.

Similarly, when you used `myVariable` in the scriptlets embedded in `First.jsp` and `Second.jsp`, the servlet container retrieves the variable by calling `getAttribute("myVariable")` on the appropriate object.

So, if you *somehow* manage to get an instance of either `HttpServletRequest` or `HttpSession`, you could stuff them with objects that could in principle be retrieved from your JSP pages. In fact, all you need is an instance of `HttpServletRequest`, since you could get the associated `HttpSession` instance by calling `getSession()` on `HttpServletRequest`.

The "somehow" will become obvious in later chapters. For now, note that this is one way you could pass data between the Java classes and JSPs that make up your webapp.

Summary

- Scopes describe the lifetime and visibility of a variable.
- **Session** and **request** scopes are the most often used scopes.
- Session-scoped variables exist for the duration of a user's session with the system.
- Request-scoped variables exist only for a single-page request, including redirects.
- Session-scoped variables are stored in an instance of `HttpSession`.
- Request-scoped variables are stored in an instance of `HttpServletRequest`.
- `HttpServletRequest` and `HttpSession` can be used to pass data between Java classes and JSPs.

CHAPTER 4

Custom Tags

Custom tags are JSP tags you create that allow you to define new functionality for your JSPs. For example, suppose we have a custom tag called `<temp:F>` that converts degrees Celsius to degrees Fahrenheit. We could use it in our JSPs:

```
Water boils at 100 Celsius, which is <temp:F>100</temp:F> Fahrenheit.
```

which displays as

```
Water boils at 100 Celsius, which is 212 Fahrenheit.
```

While this is a simple example meant to illustrate what custom tags are, in reality, custom tags are useful for a number of reasons:

- They promote code reuse because they cut down duplication of code on your JSPs.
- They make your JSPs more maintainable, because you can avoid using scriptlets.
- Their usage simplifies the work of web designers, because they obviate the need for scriptlets, which are more difficult to write and use.
- They are a key tool used to promote a separation of concerns (see Chapter 1). Most, if not all, web application frameworks, including Struts and JavaServer Faces, utilize custom tags for this reason.

Because Struts uses custom tags extensively, it is to your advantage to understand how to create and use custom tags.

Custom Tag Basics

Tags have a **prefix** defined on the JSP page (in the preceding example, the prefix is `temp`) and a **tag name** (in the example, `F`), which is fixed in the tag's **TLD** (**T**ag **L**ibrary **D**escriptor) file.

Before they can be used on a JSP, a tag has to be declared at the start of the page. For example, a declaration for the `<temp:F>` tag might be as follows:

```
<%@ taglib uri="/tags/temperature" prefix="temp" %>
```

We'll deconstruct this declaration in the following section. For now, simply note that the prefix is defined in the JSP.

■ **Note** A tag's prefix is used to prevent tag name clashes. For example, if you had to use two different tags (from different tag libraries) with the same name, you could give them different prefixes, to distinguish between them. So, prefixes are flexible and you define them on an individual JSP. Tag names, on the other hand, can't be changed and are defined in the tag's TLD file.

Tags may also have attributes, for example:

```
<msg:echo message="Hi" />
```

Lastly, several tags may be collected together, into a TLD file. That's why it's called a Tag **Library** Descriptor file. We next take a look at the lifecycle of a typical tag.

How Custom Tags Are Processed

Suppose we have a JSP with a custom tag on it. How does it get converted to text we can see on screen? Well, when the servlet container parses the JSP for the first time, it encounters the custom tag, for example:

```
<bean:write property="companyName"/>
```

The servlet container expects, and looks for, a corresponding **taglib declaration** on the JSP page. For example:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
```

There may be several such taglib declarations on the JSP page. The servlet container knows which to use by matching the prefix on the tag with the `prefix` attribute in the taglib declaration, as shown in Listing 4-1.

Listing 4-1. *A Sample JSP with a Taglib Declaration and a Custom Tag*

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
   <bean:write property="companyName"/>
```

The taglib declaration simply associates a prefix with a URI (**U**niform **R**esource **I**dentifier), which is a *logical* path to the tag's TLD file. The `uri` attribute is a logical and not a real path because it points to a specific entry in `web.xml`, the standard servlet configuration file, and not some location on your hard drive.

The web.xml file must contain the actual location of the tag library indicated by the uri attribute. In this case, the relevant information in web.xml is given in a <taglib> tag (see Listing 4-2).

Listing 4-2. *A Taglib Section in web.xml*

```
<taglib>
   <taglib-uri>/tags/struts-bean</taglib-uri>
   <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
```

The main points to notice from Listing 4-2 are

- The taglib-uri body must match the uri attribute of the taglib declaration given in the JSP file.
- The taglib-location body must contain the relative path to the TLD file. In the previous example, the TLD file is struts-bean.tld.

The physical location of the TLD file is always relative to the root folder of the webapp's WAR file (or relative to your web application's subdirectory in the servlet container's webapps directory). So, in Listing 4-2, we see that the TLD file resides in the WEB-INF folder and is called struts-bean.tld.

Finally, the TLD file is just an XML document containing metadata describing the tags in the library. This metadata is

- A tag's name (which in Listing 4-1 would be write).
- A tag's optional and required attributes and (property from Listing 4-1).
- The Java handler class that processes the tag's attributes and body to produce the output. These Java classes are usually bundled in a JAR file that you place in your webapp's WEB-INF\lib directory.

We'll go through the nitty-gritty details of the TLD file in the lab session. Figure 4-1 summarizes the processing flow of a custom tag.

In the final step, the servlet container calls certain functions on the Java tag handler, which is responsible for producing the custom tag's output.

As you can see, the processing flow of even the simplest custom tag is a little involved. I suggest rereading this section a couple of times until you are reasonably confident of the details before proceeding.

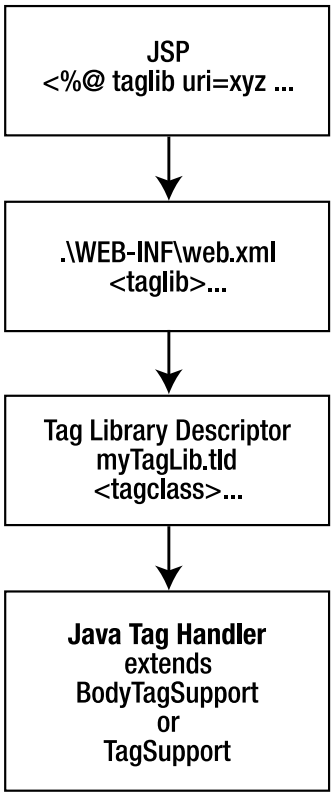

Figure 4-1. *Summarized processing flow of a custom tag*

The Java Handler Classes

The actual work of transforming the custom tag's body and attributes into HTML code is done by Java handler classes that you must implement.

Your handler classes would subclass one of two base classes, depending on the requirements of your tag.

If your custom tag has **no body**, then you'd subclass `javax.servlet.jsp.tagext.TagSupport`. This base handler class represents custom tags having no body, but possibly having attributes.

If your custom class has a body, then you'd have to subclass `javax.servlet.jsp.tagext.BodyTagSupport` instead. Of course, using `BodyTagSupport` doesn't mean your tags must have a body when they are used on JSPs. What it does mean is that you have to implement an extra handler function to process the tag's body.

To keep things simple, in what follows we will focus on `BodyTagSupport`.

■ **Note** I've bent the truth a little here. You can subclass either `BodyTagSupport` or `TagSupport`, regardless of whether or not your custom tag has a body. It's just that `BodyTagSupport` has extra facilities to allow you to conveniently read your tag's body. `TagSupport` does not have this ability. But this convenience comes at a price, since the servlet container has to do extra work and use more memory. This explains why you should subclass `TagSupport` if your custom tag doesn't have a body, and use `BodyTagSupport` only if necessary.

That extra function is `doAfterBody()`, which is called after the servlet container reads in the tag's body. This function is where you read the tag's body, do all necessary processing, and display the output. The tag must return an integer flag called `EVAL_PAGE`, defined for you on `BodyTagSupport`. This flag tells the servlet container to go ahead and process the rest of the page.

The other requirement your handler classes must satisfy is that they must have additional `getXXX` and `setXXX` functions corresponding to attributes supported by your custom tag.

■ **Note** In other words, in addition to being a subclass of either `BodyTagSupport` or `TagSupport`, your handler class needs to be a JavaBean, with properties corresponding to attributes supported by your custom tag.

For example, in Listing 4-1, by looking at the `<bean:write>` you can infer that the corresponding Java handler class has the functions `getProperty()` and `setProperty()`. Listing 4-3 illustrates this.

Listing 4-3. *Part of the Java Handler Class for the <bean:write> Tag*

```
import  javax.servlet.jsp.tagext.*;
public class BeanWriteTagHandler extends TagSupport{

   protected String _property = null;

   public String getProperty(){
      return _property;
   }

   public void setProperty(String property){
      _property = property;
   }
```

```
   //other code omitted.
   ...
}
```

When the JSP that contains your tag loads, the servlet container calls all the necessary `setXXX()` functions so that attributes of your tag are accessible to your tag's Java code. Using the `<bean:write>` tag as an example again, at runtime, you could call the `getProperty()` function or just look at `_property` to read the value of the `property` attribute of the `<bean:write>` tag.

Helper Classes

In order for your Java handler classes to do anything useful, such as reading the contents of a tag's body or writing your tag's output, there are a couple of helper classes you need to know about.

The first is `javax.servlet.jsp.tagext.BodyContent`, which you use to read your tag's body, among other things. The function `getString()` returns your custom tag's body (assuming it has one) as a `String`. You can get an instance of `BodyContent`, if you've subclassed `BodyTagSupport`, by calling the function that has been implemented for you on the base class `BodyTagSupport`.

`BodyContent` also gives a `Writer` instance with which you can write the output of your tag's transformed body. Calling `getEnclosingWriter()` on the `BodyContent` instance will give you a `Writer` subclass—specifically, a subclass of `javax.servlet.jsp.JSPWriter`.

To see how all this works, consider a simple `<message:write>` tag, used like this:

```
<message:write font-color="red">Hello World!</message:write>
```

We need the message "Hello World!" to be displayed in the given font color—red, in this instance. A Java handler to accomplish this is shown in Listing 4-4.

Listing 4-4. *Implementation of <message:write>'s Handler Class*

```
package net.thinksquared.tags;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
public class MessageWriteTagHandler extends BodyTagSupport{

   protected String _fontColor = "black"; //default font color

   public String getFontColor(){
      return _fontColor;
   }
```

```
   public void setFontColor(String fontColor){
      _fontColor = fontColor;
   }

   public int doAfterBody(){

      try{

        BodyContent bc = getBodyContent();
        JSPWriter out = bc.getEnclosingWriter();

        out.write("<font color=\"");
        out.write(_fontColor);
        out.write("/">");
        out.write(bc.getString());
        out.write("</font>");

      }catch(Exception ignore){}

      //tell the servlet container to continue
      //processing the rest of the page:
      return EVAL_PAGE;
   }

}!
```

The TLD File

The last piece of the custom tag puzzle is the TLD (**T**ag **L**ibrary **D**escriptor) file. Each tag you create has to be declared in a TLD file, and this file has to be deployed along with the tag's Java handler classes. As we've seen in the previous section, the servlet container knows where you've placed the TLD file because you've declared the path in `web.xml`, the standard servlet configuration file.

■ **Note** I've included this section mainly for your reference. If you're itching to write your own custom tag, you may safely skip this section and proceed to Lab 4.

Consider the TLD file declaring the `<message:write>` tag we developed earlier, as shown in Listing 4-5.

Listing 4-5. *TLD File Declaring <message:write>*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib
   PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
   "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
    <tlibversion>0.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>message</shortname>

    <tag>
        <name>write</name>
        <tagclass>
           net.thinksquared.tags.MessageWriteTagHandler
        </tagclass>
        <bodycontent>JSP</bodycontent>
        <attribute>
            <name>font-color</name>
            <required>false</required>
        </attribute>
    </tag>

</taglib>
```

As you can see, the TLD file is just an XML file. The root tag is `<taglib>`, and this contains one or more `<tag>` tags, which declare your custom tag. Tables 4-1, 4-2, and 4-3 explain each tag on the TLD file. Note that each `<taglib>` must contain one or more `<tag>` declarations.

Table 4-1. *Tags on the TLD File*

Tag	Meaning
`taglib`	Root tag for the TLD file.
`tlibversion`	Your version number for this tag library.
`jspversion`	The JSP version this tag library depends on. Use `1.1`.
`shortname`	The preferred or suggested prefix for tags on this tag library, when you use the tag on your JSP pages. You are of course free to ignore this suggestion.
`tag`	Contains metadata for a single tag in this tag library.

Table 4-2. *The Subtags of <tag>*

Tag	Meaning
`name`	Name of this tag.
`tagclass`	Name of the Java handler class. Note that this is the fully qualified name of the handler class, e.g., `net.thinksquared.tags.MessageWriteTagHandler`.
`bodycontent`	The type of content for this tag's body. Use `JSP`.
`attribute`	Contains metadata for a single attribute on this tag.

Table 4-3. *The Subtags of <attribute>*

Tag	Meaning
`name`	Name of this attribute. This obviously has to be unique on the tag.
`required`	Indicates if this tag is required (`true`) or optional (`false`).

Lab 4: A Temperature Conversion Tag

In this lab session, you will create, deploy, and test a temperature conversion tag. Here are some specs:

- The tag library will contain just one tag called `convert`, which has one optional attribute.
- The optional attribute `to` indicates the temperature scale to convert to.
- By default, `to` is Fahrenheit.
- Conversions are from Celsius to either Fahrenheit or Kelvin.

■ **Note** In a production implementation, you'd include some error reporting.

For example, these all convert to 100 Celsius to 212 Fahrenheit:

```
<temp:convert to="Fahrenheit">100</temp:convert>
<temp:convert to="F">100</temp:convert>
<temp:convert>100</temp:convert>
```

while these convert 100 Celsius to 373 Kelvin:

```
<temp:convert to="Kelvin">100</temp:convert>
<temp:convert to="K">100</temp:convert>
```

To create and use this tag, you will have to

1. Prepare the development directory and scripts.
2. Write the Java tag handler.
3. Write the TLD file describing the tag.
4. Write the JSP to test the custom tag.
5. Amend `web.xml` to register your TLD file.
6. Install the application onto Tomcat.

Step 1: Prepare the Development Environment and Scripts

1. Create a `Struts` directory in a convenient location on your hard drive.
2. Copy the `lab4.zip` file on the CD-ROM into this `Struts` directory.
3. Unzip the contents, making sure you've preserved the directory structure.
4. You should see a subdirectory called `.\Struts\lab4\`.
5. Open `.\Struts\lab4\compile.bat` in a text editor and amend the `PATH` environment variable so that it points to your JDK installation.

■ **Note** On Windows XP, ME, or 2000, which have built-in unzipping capabilities, the zip file may add an extra `lab4` directory in the unzipped path. So, the `compile.bat` file's path would be `.\Struts\lab4\lab4\compile.bat`. You could move the `lab4` folder up or just continue. The compile scripts should work regardless.

Test out your changes by clicking `compile.bat`. You should get no errors, and you should see a file called `lab4.war` in `.\Struts\lab4\`.

In what follows, I'll refer to all paths relative to `.\Struts\lab4\`.

Step 2: Write the Java Tag Handler

1. Open the file `.\src\Converter.java` in your favorite text editor.
2. Put in a private `String` variable called `_to`. This will correspond to the `to` attribute of your custom tag.
3. Create `getTo()` and `setTo()` methods to get and set the value of `_to`. The servlet container will use these to get/set the variable `_to` with the value of the `to` attribute.
4. Complete the `doAfterBody()` method, using the specs to guide you. You will need to use helper classes to do this (see Listing 4-4).
5. Note that `doAfterBody()` must return the integer flag `EVAL_PAGE` to indicate that the rest of the JSP page is to be evaluated.
6. Compile your work by clicking on `compile.bat`.

Step 3: Writing the Tag Library Descriptor file

1. Open `.\web\WEB-INF\lab4-converter.tld` in your favorite text editor.
2. This is an empty tag library descriptor file, containing just the mandatory boilerplate. Create the root `<taglib> ... </taglib>` tag after the boilerplate.
3. Within the enclosing `<taglib>` tag, insert the appropriate tags, using Listing 4-5 as a reference.

Step 4: Amend web.xml

`web.xml` is the standard servlet configuration file. Every webapp must have its own `web.xml` file, even if it's a blank one.

■ **Note** If you place your TLD files in the `WEB-INF` directory, you really don't need to declare them in `web.xml`. We will, however, for completeness.

---

1. Open `.\web\WEB-INF\web.xml` in your favorite text editor. Note that the `web` directory exists only in development. The `compile.bat` script later moves the whole `WEB-INF` directory up, and removes the `web` folder.

2. The `web.xml` file contains boilerplate text, followed by a `<webapp>` tag. Insert the tags shown in Listing 4-6 within the enclosing `<webapp>` tag.

Listing 4-6. *Your Addition to web.xml*

```
<taglib>
   <taglib-uri>/tags/lab4-converter</taglib-uri>
   <taglib-location>/WEB-INF/lab4-converter.tld</taglib-location>
</taglib>
```

■ **Note** Notice in Listing 4-6 that the path separators are Unix-style slashes (/), not Windows backslashes.

Step 5: Write Your JSP

1. Open the file `.\web\test.jsp` in your favorite text editor.
2. Put in the taglib declaration for your custom tag library. Remember to use the URI you defined in `web.xml`.
3. Put in JSP code to test the convert tag. You should at least test all the examples given at the start of this lab.

Step 6: Deploy and Test

1. Shut down Tomcat if it is running.
2. Click `compile.bat`. This compiles your source code and produces the WAR file, `lab4.war`.
3. Drop the WAR file into Tomcat's `webapps` directory. Remember, if you want to re-deploy the `lab4` app, you will have to delete its folder under the `webapps` directory.
4. Use `http://localhost:8080/lab4/` to test your work.

If you're stuck at any point, you might want to consult the answers to this lab. You'll find them in `lab4-answers.zip` on the CD-ROM.

Professional Java Tools

In this book, I've used Windows batch files to compile and prepare WAR files. I've done this to keep things as simple as possible. However, in the real world, batch files are not the best way to compile your web applications. A much, much better option is to use Apache Ant (see "Useful Links" for the website).

Ant gives you a platform-independent way of compiling your Java source code. But Ant does much more. You can create different compile configurations easily with Ant. Also, many valuable Java tools integrate with Ant. While using Ant is beyond the scope of this book, I encourage you to download it and try to use it in your projects.

The other area I encourage you to explore is using a real Java integrated development environment (IDE), rather than plain ol' Notepad. Most IDEs provide tools to simplify creating and editing Java, HTML, and XML, thus making you more productive.

One IDE I highly recommend is the open source Eclipse (see "Useful Links" for the website). A copy of this IDE is on the accompanying CD-ROM.

Useful Links

- Apache Ant website: `http://ant.apache.org`
- Eclipse website: `www.eclipse.org`

Summary

- Custom tags are a central piece in Java-based web application frameworks.
- Custom tag functionality is implemented by your subclass of either `TagSupport` or `BodyTagSupport`.
- Use `TagSupport` if your custom tag doesn't have a body and `BodyTagSupport` if it does.
- You need to declare your custom tag in a Tag Library Descriptor (TLD) file.
- You might have to register the TLD file's location in `web.xml`.

CHAPTER 5

The MVC Design Pattern

Servlet technology is the primary base on which you can create web applications with Java. JavaServer Pages (JSP) technology is based on servlet technology but extends it by allowing HTML content to be created easily.

Note that JSP technology doesn't *replace* servlet technology, but rather *builds on* it, to address some of its deficiencies. Struts follows this pattern, and builds on servlet and JSP technologies to address their shortcomings. These shortcomings are in two broad areas:

- **Creating a "separation of concerns"**: Pieces of code that do different things are bundled separately and given standardized ways of communicating between themselves.

- **An infrastructure for webapps**: This includes things like validating user input, handling and reporting errors, and managing flow control.

If these shortcomings are addressed, the very practical problems of building webapps (robustness, maintainability, localization, etc.) become easier to tackle.

One tool that can be used to make some headway in resolving these issues is an organizing principle called the **M**odel-**V**iew-**C**ontroller (MVC) design pattern.

WHAT ARE DESIGN PATTERNS?

Design patterns are essentially recipes for solving generic coding problems. They are not algorithms but rather *organizational principles* for software.

The idea of using "design patterns" to solve common software engineering problems first came to prominence with the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional, 1995), by, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The authors credit an earlier work, *A Pattern Language for Building and Construction* (Oxford University Press, 1977) by the architect Christopher Alexander as the inspiration for the idea of design patterns for software.

The MVC design pattern, which is the subject of this chapter, was first used in an AI language called Smalltalk in the 1980s to solve a problem similar to the one we're discussing here for webapps. A variation on the pure MVC pattern is also used in Swing, Java's platform-independent GUI toolkit.

The MVC design pattern calls for a separation of code by their function:

- **Model**: Code to control data access and persistence
- **View**: Code to handle how data is presented to the user
- **Controller**: Code to handle data flow and transformation between Model and View

Figure 5-1 illustrates MVC.

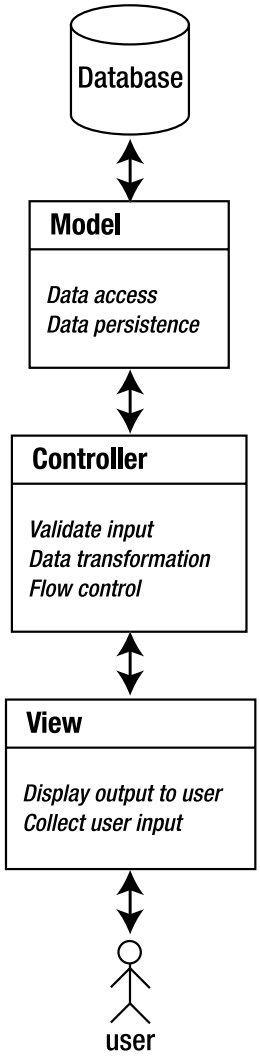

Figure 5-1. *The MVC design pattern*

In addition to this separation, there are a couple of implicit constraints that the Model, View, and Controller code follow:

- The View gets data from the Model only through the Controller.
- The Controller communicates with the View and Model through well-defined, preferably standardized ways. For example, embedding SQL code directly in your Controller code violates this principle of standardized communication. Using Model classes that expose functions for data access would be an example of standardized communication between the Controller and the Model.

It may not be immediately obvious to you how this solves anything, but I hope you'll come to appreciate the power of the MVC approach as you progress. For now, it is sufficient for you to accept that applying the MVC design pattern helps us achieve the larger goal of separation of concerns, which greatly simplifies the building of web applications.

Our discussion so far has been quite general for a purpose. *Any* web application framework will use the MVC principle, so it is to your advantage to understand it apart from Struts' implementation of it.

Later, we'll discuss how Struts implements the MVC design pattern. But before we do, I'll give you an example of how to apply the MVC design pattern.

The Registration Webapp

Many web applications have a "user registration" facility. I'm sure you've seen this before. Some online news sites, for example, have the annoying requirement that you "subscribe" before you can view their content.

I'll call this facility the "Registration Webapp." What I'll do is specify the requirements of this webapp, and for each requirement, I'll identify the types of code (Model, View, or Controller) that might be needed.

Requirement 1

The user is asked to specify a user ID (userid) and a password, as well as a password confirmation, as shown in Figure 5-2.

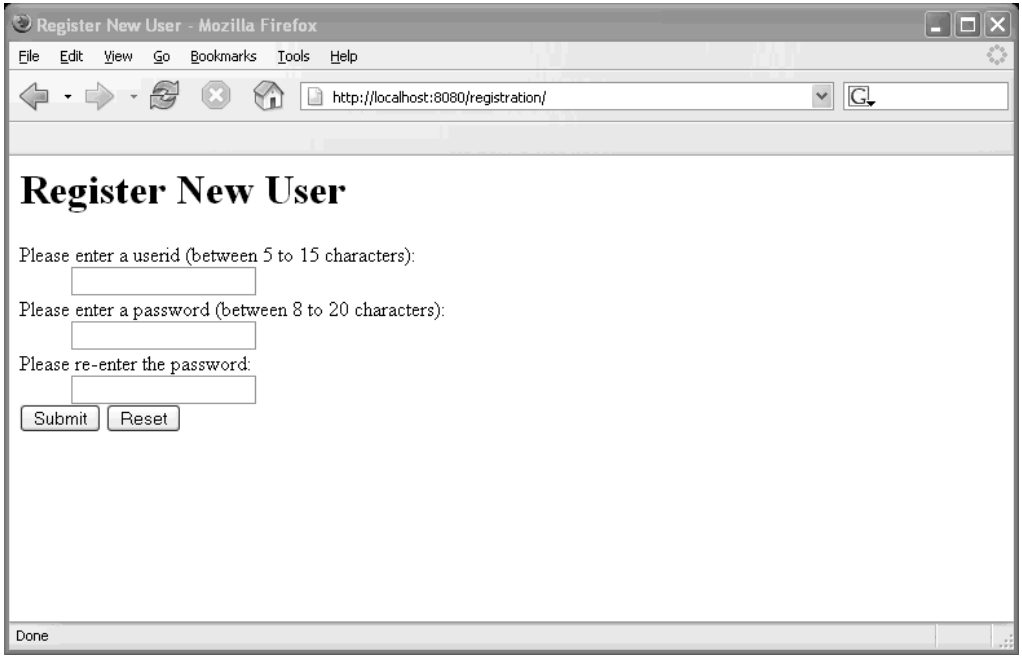
Register New User - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/registration/
Register New User
Please enter a userid (between 5 to 15 characters):
Please enter a password (between 8 to 20 characters):
Please re-enter the password:
Submit
Reset
Done

Figure 5-2. *The Registration webapp main page*

Discussion

Clearly, you need View code to create the form containing the user ID and password fields. The output of the View code would be HTML. The View code itself might be this displayed HTML, or a JSP that is transformed into HTML by the servlet container.

Requirement 2

The form data is validated to check for a malformed user ID or password. The form is also checked to ensure that the same password was entered both times and that it has the right length.

Discussion

You will need Controller code to do the validation. One important point to note is that these checks can be done without referring to a database. This means that you could validate either on the client (the web browser), perhaps using JavaScript, or use Java classes residing on the server.

The first option would bend the MVC rule of strict separation between View and Controller because the JavaScript would have to be placed in the HTML or JSP code in Requirement 1. Even if you factored out the JavaScript validators into a separate `.js` file, you'd still need Controller code on the HTML/JSP View code to transfer the form values to the JavaScript validators. Don't get me wrong; I'm *not* saying that client-side JavaScript is an evil that crawls out from the bowels of Hell. I'm simply saying that *hand-coded* client-side JavaScript to perform validation violates MVC.

Violations of MVC, *no matter how well justified*, come at the cost of reduced maintainability.

Fortunately, it is possible to have your cake and eat it too. Instead of using hand-coded client-side JavaScript, Struts gives you the option of *autogenerating* client-side JavaScript. This would greatly reduce Controller/View overlap. You'll see this in action when we cover the Validator framework in Chapter 15.

Or, we could simply cut the Gordian knot and do all validations on the server. Unfortunately, this isn't always an option, especially if the client-server connection is slow.

Requirement 3

The data is further checked to ensure that there are no other users with that user ID.

Discussion

As in Requirement 2, we obviously need Controller code to run the validation. Unlike Requirement 2, a client-side validation is less feasible, because we might need to check with a database in order to find out if the given user ID exists.

This also implies we need Model code to read user IDs off the database. The Model code has a function like this:

```
public boolean exists(String userid)
```

to run the user ID check. Our Controller code would use this function to run the validation.

Requirement 4

If at any stage there is an error, the user is presented with the same start page, with an appropriate error message (see Figure 5-3). Otherwise, the user's user ID and password are stored.

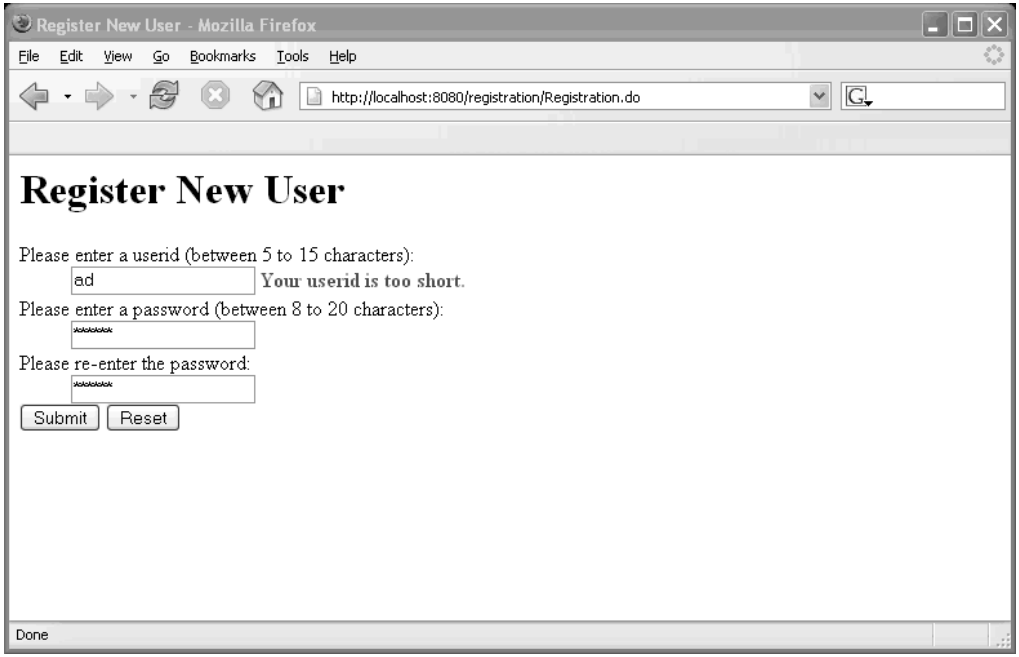
Register New User - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/registration/Registration.do
Register New User
Please enter a userid (between 5 to 15 characters):
ad
Your userid is too short.
Please enter a password (between 8 to 20 characters):
Please re-enter the password:
Submit
Reset
Done

Figure 5-3. *Signaling errors to the user*

Discussion

You'll need a Controller to decide what to do if there's an error: display the error page or save the data?

You'll also need a View to display the error message. Here's where enforcing strict MVC separation makes things complicated. For example, do you need to duplicate View code from Requirement 1 to redisplay the form data? How about re-populating the previously keyed-in form data? In Figure 5-3, the user ID field isn't blank—it contains the previously keyed-in user ID.

In fact, we could handle errors from Requirement 2 if we threw out MVC and mixed Controller and View code, as I described in the discussion to Requirement 2. Although this still wouldn't solve the same display issues arising from Requirement 3, we might be able to hack a solution by merging Model and Controller code with our View code by embedding scriptlets in our JSP View code.

These complications illustrate the need for a web application framework like Struts. You need Struts to help you enforce MVC. Without Struts (or a web application framework), it is probably impossible to cleanly enforce MVC separation on your webapp code.

Note that I'm only saying that Struts (or a webapp framework) makes MVC *possible*. Don't take this to mean that Struts makes bad code impossible! You can cheerfully violate MVC even if you're using Struts.

Lastly, you'd also need Model code to save the user ID and password pair. Listing 5-1 outlines how the Java class for your Model code would look.

Listing 5-1. *Skeleton for the Registration Webapp Model*

```
public class User{

   protected String _userId = null;
   protected String _password = null;

   /* Bean-like getters and setters */
   public String getUserId(){
      return _userId;
   }

   public String setUserId(String userId){
      _userId = userId;
   }

   public String getPassword(){
      return _password;
   }

   public String setPassword(String password){
      _password = password;
   }

   /**
   * Checks if the userid exists.
   */
   public static boolean exists(String userid){
      //implementation omitted
   }

   /**
   * Saves the _userid and _password pair
   */
   public void save() throws Exception{
      //implementation omitted
   }
}
```

Of course, in real applications, you'd have more than one class for your Model code.

Requirement 5

If the registration is successful, the new user gets a "You are registered" web page (see Figure 5-4).

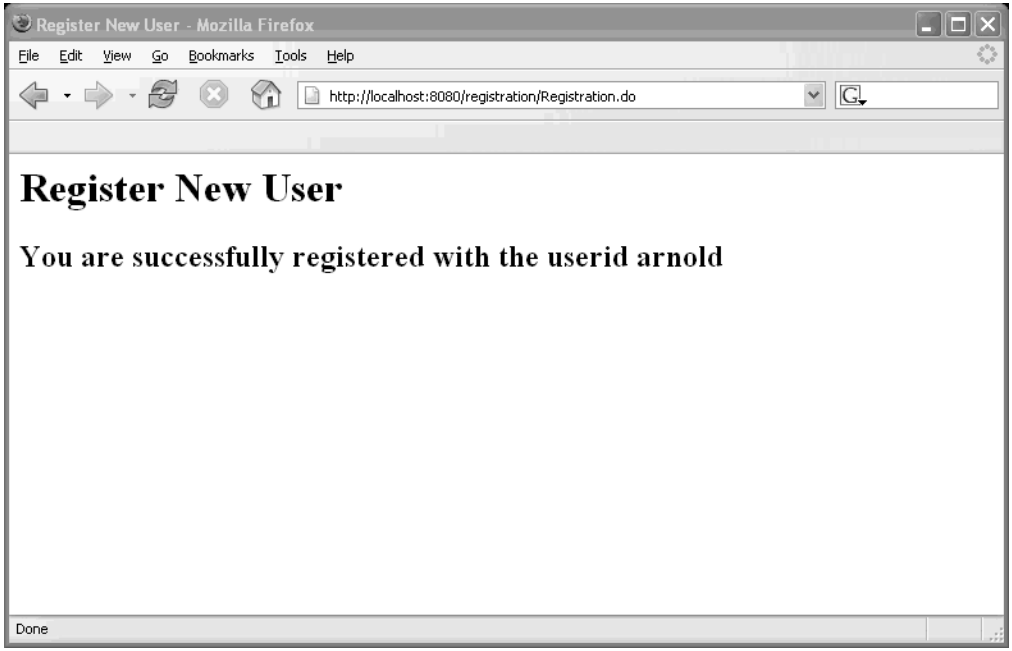
Register New User - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/registration/Registration.do
Register New User
You are successfully registered with the userid arnold
Done

Figure 5-4. *Successful registration*

Discussion

You'll need View code to display the page. Plain ol' HTML would do, unless you're embedding this message within a larger page, as is often the case in real life. Like Requirement 4, this latter possibility also complicates strict MVC enforcement.

But again, Struts comes to your rescue. As you'll see in Part 2 of this book, Struts has an extension ("plug-in" in Struts-speak) to give you just this functionality.

I hope this simple example gives you a good idea how to classify your code with the MVC worldview. I also hope it gives you an idea of why you'd need a framework to help you actually enforce MVC. To give you a little more practice, try the following quiz.

Lab 5: MVC Quiz

LILLDEP (Little Data Entry Program) is a simple data entry program for storing and managing a database of contacts. It's a common type of web application used in many SMEs (**S**mall **M**edium **E**nterprises). Figure 5-5 shows the basic LILLDEP main page.

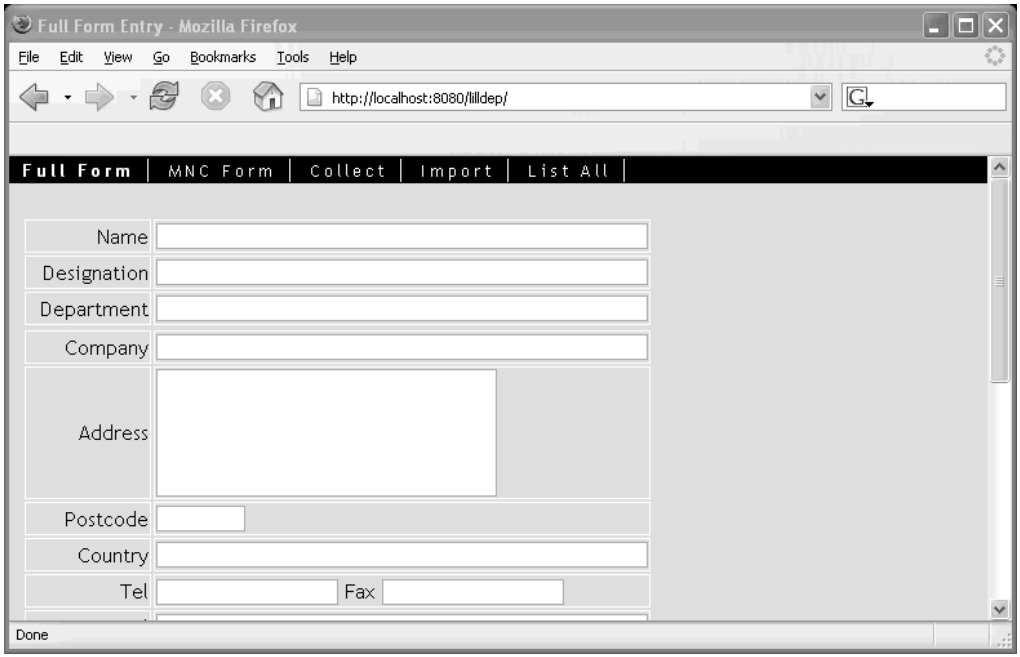

Figure 5-5. *The LILLDEP main page*

Have a look at the three requirements that follow and try to identify the Model-View-Controller parts that might be relevant. Imagine a possible implementation for each, as I did in the previous section, and identify where you might need help to enforce strict MVC code separation.

- **Requirement 1**: A page to collect information about a contact
- **Requirement 2**: Code to check contact details (postal code, email, or required fields) after submission and display errors
- **Requirement 3**: Code to store and retrieve data into a database

Which Comes First?

When you develop your webapp with the MVC design pattern, which portion (Model/View/Controller) would you specify and design first? Why?

Close the book and give yourself a few minutes to answer this before reading the following discussion.

There's no "right" answer to this one, but most people start with either View code or Model code. Rarely would you start with a design for the Controller since it depends on View and Model components.

My personal preference is to specify and design Model code first. This means defining at an early stage *what* data to store and *how* to expose data with data access methods (like the `exists()` method in Listing 5-1) to other code. Here are my reasons for holding this position:

- Without a clear specification of *what* data to store, you can't design the rest of the application.

- Without a specification of *how* to access data, a clear separation between Model and Controller is difficult to achieve. For example, if you're using a relational database, you might end up with SQL code in the Controller. This violates the MVC principle of well-defined interfaces.

Many people (especially new developers) prefer to design View code first, possibly because this helps them understand the application's inputs and control flow. In this approach, View comes first, then Controller, and Model last. Though it's perfectly valid, I believe this approach is prone to certain pitfalls:

- **Redundant information** being stored in the database. Like the proverbial three blind men describing the elephant, using View code to drive Model design is bad because you lack a global view of the application's data requirements. This is especially so in complex applications, where you could store the same information twice, in two different tables.

- **Poorly organized database tables**, because the Model code and data organization is designed to conform to View requirements, which may not reflect the natural relationships between Model elements.

- **Less future-proof code,** because a change in the View code may necessitate a revamp of Model code and the database design. Essentially, Model code is too strongly coupled to View code.

All these stem from the fact that those who take the "View first" approach use View code or mockups to indirectly educate them about Model requirements.

Having said that, there are many situations where a "Model first" approach may *seem* impractical. For example, in many corporate IT departments, developers don't have a full understanding of the problem domain. Instead, they have to rely on and frequently communicate with non-IT personnel like managers or admin staff ("domain experts") in order to understand the system's requirements.

In this scenario, the application is naturally View driven because mockups are used as a communication tool between developers and domain experts. My advice to those in such a situation is to *incrementally* build a picture of the Model code and data immediately after each discussion with domain experts. Don't leave Model design to the last.

Whichever way you choose, Model-first or View-first, I believe it's important to understand why you do things the way you do, simply because this helps you become a better programmer.

In the following section, I'll describe how Struts fits into the MVC design pattern.

Struts and MVC

In Struts, **Model** code consists of plain old Java objects (POJOs).

In other words, Struts places no restrictions on how you code the Model portion of your webapp. You should encapsulate data access/persistence into Model classes, but Struts does not force you to do this. Remember, Struts is here to help you achieve MVC nirvana as easily as possible. Apparently, you don't really need help to separate Model from View and Controller.

View code consists of JSPs and a set of custom tags supplied by Struts. These custom tags enable you to separate View from Controller, because they address all the issues I raised earlier when I tried to apply MVC to the Registration webapp.

With Struts, **Controller** code falls into three broad categories:

- **Simple validations** are performed by your subclasses of the Struts base class called `ActionForm`. Checks for password length or email address format are examples of this. Later in this book, you'll see how Struts greatly simplifies validation with the Validator framework.

- **Complex validations and business logic** are done in your subclasses of the Struts base class called `Action`. The check for the duplicate user ID for the Registration webapp is an example of complex validation. An example of business logic is calculating the total amount due after a purchase in a shopping cart webapp.

- **Flow control** is also decided by your `Action` subclasses, restricted to paths declared in the Struts configuration file called `struts-config.xml`.

You'll learn more about `ActionForm`, `Action`, and `struts-config.xml` as this book progresses. At this stage, you should simply understand how MVC maps into Struts. Figure 5-6 depicts this mapping.

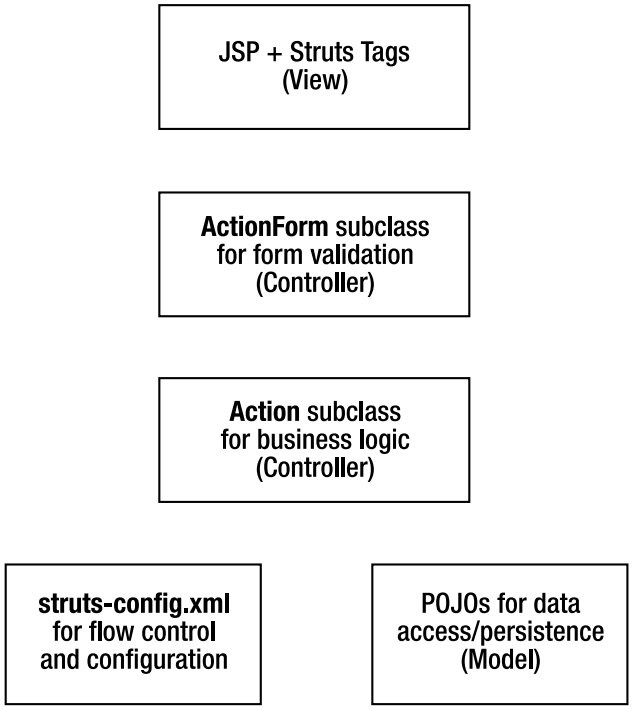

Figure 5-6. *How MVC maps into Struts*

Lifecycle of a Struts Request

The three categories of processing (simple/complex validation, business logic, and flow control) come into play when an incoming HTTP request is generated by a user submitting form data.

With servlet technology all such processing is done by servlets. As I mentioned in Chapter 2, Servlets are just Java objects, *usually* your subclasses of `HttpServlet`.

Even JSP pages are converted at runtime into servlet classes—actual `.java` source files. These autogenerated Java source files are later compiled.

Since Struts is built on servlet technology, it is no exception to this rule. All submissions of form data to a Struts application are intercepted by a *single* "master" servlet, an instance of `ActionServlet`, which is a part of Struts.

This master servlet is responsible for delegating the actual work of your application to your subclasses of `ActionForm` and `Action` (see Figure 5-6).

USUALLY?

Servlet technology is quite generic, meaning it is not at all tied to processing HTTP requests, although this is by far the most common use.

This is because servlets solve a number of *generic* problems that stem from client-server communication, such as object pooling and session management. Your servlet classes could implement the basic `Servlet` interface from scratch, or subclass the `GenericServlet` base class. You'd do this if you wanted to take advantage of servlet technology's solutions to these generic problems.

If you're primarily concerned with processing and displaying web pages, `HttpServlet` is the base class you'd subclass.

■ **Note** In addition to this, `ActionServlet` and its helper classed are responsible for many other behind-the-scenes things like pooling `Action` subclasses for efficiency, reading configuration data, or initializing Struts extensions called plug-ins. These activities are invisible to Struts developers, and are one good reason why Struts is so easy to use. You don't have to know how the clock works to tell the time!

The incoming form data is processed in stages:

- **Data transfer**: Form data is transferred to your `ActionForm` subclass.
- **Simple validation**: The form data in this subclass is passed through simple validation. If simple validation fails, then the user is automatically presented with the form with the error messages detailing the errors. This is a *typical* scenario, but the details would depend on how you configured Struts. We'll go through the mechanics of simple validation in Chapter 6 and tackle configuration in Chapter 9.
- **Further processing**: Form data is next sent to your `Action` subclass for complex validation and the processing of business logic. We'll discuss this in Chapter 7.

Your `Action` subclass also specifies the "next" page to be displayed, and this ends the processing of a single request.

Frameworks for the Model

In a previous section, I sang the joys of designing the Model portions of your webapp first, so Struts' lack in this area might come as a bit of a letdown.

One reason for this glaring deficiency in Struts is because there are already several frameworks to help you with data access and persistence. These go under the monikers "persistence framework/layer" or "object/relational (O/R) persistence service." The reason these frameworks exist is to persist data, and do it *well*. You may use any one of these in conjunction with Struts. So you see, it isn't a deficiency at all but just good design.

There are two different approaches to data persistence:

- **Persist the classes**: This approach lets you store and retrieve any Java object to/from a relational database. You are provided with a set of classes, which you must use to persist and retrieve your own Java classes. An example of this approach is Hibernate, from Apache.

- **Classes that persist**: In this approach, you specify a description for your database tables and their interrelationships. The persistence framework *autogenerates* Java classes (source code), which have mappings to your database tables. These autogenerated classes have `save()`, `select()`, and `delete()` functions to allow persistence and retrieval of data objects. A prime example of this approach is Torque, also from Apache.

The "persist the classes" approach is by far the more flexible of the two, since you could potentially persist any Java class, including legacy ones, or third-party classes. The "classes that persist" approach, on the other hand, lacks this capability but produces a cleaner separation between Model and Controller.

I should warn you that while this comparison between approaches is valid, you can't use it to choose between products like Hibernate and Torque. Hibernate, for example, provides many features beyond what I've described here. These extra features are what you'd want to consider when choosing a persistence framework.

In Appendix A, I give you simple examples of how to use Hibernate and Torque, and also (shameless plug) a simple, open source, "classes that persist" framework called Lisptorq, developed by the company I work for.

Useful Links

- Hibernate: `www.hibernate.org/`
- Torque: `http://db.apache.org/torque/`
- Lisptorq: `www.thinksquared.net/dev/lisptorq/`

Summary

- The Model-View-Controller (MVC) design pattern brings many benefits to the webapp that implements it.
- Implementing MVC is difficult, which is why you need a web application framework like Struts.
- Struts only places restrictions on the View and Controller. You are free to implement the Model portion in any way you wish.
- Two very popular persistence frameworks are Hibernate and Torque.

CHAPTER 6

Simple Validation

One common element of web applications is *form validation*. This involves running checks to ensure that user input conforms to the system's expectations. In the previous chapter we saw how these checks naturally fall into two categories:

- **Simple validation**: Validations that do not require complex processing or business logic. This includes, for example, checks that mandatory fields are filled, that user input conforms to the right format (email addresses, postal codes, or credit card numbers), and a variety of other checks.

- **Complex validation**: Validations that are dependent on processing of business logic. For example, a check for a duplicate user ID would qualify as a complex validation because it involves reading data from a Model component.

Also in the previous chapter, I briefly described how Struts fits into MVC. You also saw how for the Controller, Struts makes a distinction between "simple" and "complex" validations of user input by using different classes to perform them.

Struts makes this distinction for two reasons. First, they involve significantly different operations and are usually maintained separately. The second reason is to give you a powerful tool (the Validator framework), which greatly relieves the tedium of doing simple validation.

In this chapter, we'll focus on the basics of simple validation.

Processing Simple Validations

In addition to using different classes for simple and complex validations, Struts processes these validations differently too.

When a user submits a form, Struts first runs simple validations on it. If the validations fail, Struts sends the user the same form, with the same data the user keyed in but with error messages corresponding to each field with an error.

Only after the simple validations pass will Struts send the form data for further processing. This might include performing complex validations or data transformations or storing data into the database.

Anatomy of ActionForm

The central class for performing simple validations is

`org.apache.struts.action.ActionForm`

Struts requires you to associate each form displayed to the user with your subclass of `ActionForm`.

Your subclass of `ActionForm` does two things:

- **It holds all form data**: Your subclass of `ActionForm` must have getters and setters corresponding to each property of the form.
- **It performs simple validation**: Your subclass must override the `validate()` function, *if* you want to perform any simple checks.

Figure 6-1 illustrates this relationship between `ActionForm`, your `ActionForm` subclass, and the input HTML form for which your subclass stores data.

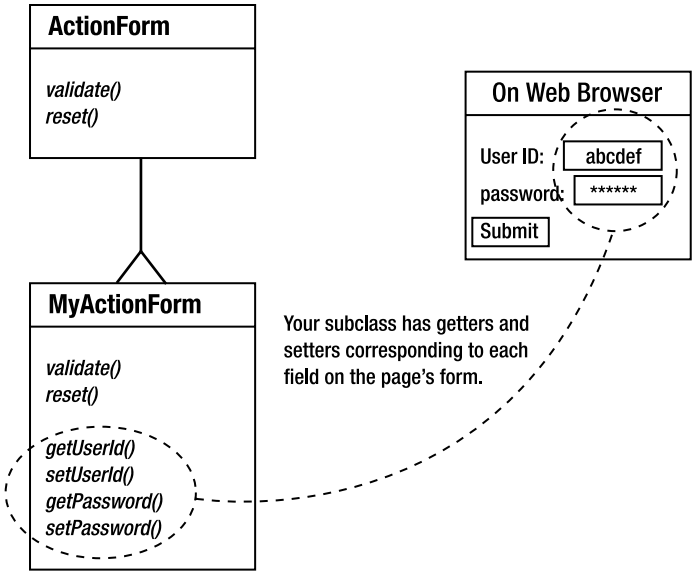

Figure 6-1. *Subclassing ActionForm*

When a user submits a form, Struts populates the associated `ActionForm` subclass with the data, then calls `validate()` to perform any simple validations required. This function looks like this:

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)
```

■ **Note** For the time being, (and for your sanity!) you may safely ignore the `ActionMapping` and `HttpServletRequest` classes, which are `validate()`'s arguments. At this stage, the only important class apart from `ActionForm` is `ActionErrors`. We'll come back to the other two classes in future chapters.

`ActionErrors`, which is `validate()`'s return value, is essentially like a `HashMap` to store error messages by key. If `validate()` returns a `null` value (or an empty `ActionErrors` instance), the validation passes.

If `validate()` returns a non-empty `ActionErrors` instance, the validation fails, and Struts redisplays the form to the user along with any error messages.

Listing 6-1 illustrates how you might implement the `ActionForm` subclass corresponding to the Registration webapp form of Chapter 5.

Listing 6-1. *RegistrationForm.java*

```
package net.thinksquared.registration.struts;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public final class RegistrationForm extends ActionForm{

   private String _userid = null;
   private String _pwd = null;
   private String _pwd2 = null;

   /**
    *   getXXX and setXXX functions
    *   corresponding to form properties
   */
   public String getUserid(){ return _userid; }
   public void setUserid(String userid){ _userid = userid; }

   public String getPassword(){ return _pwd; }
   public void setPassword(String pwd){ _pwd = pwd; }

   public String getPassword2(){ return _pwd2; }
   public void setPassword2(String pwd2){ _pwd2 = pwd2; }
```

```
    /**
     *  Validate the user input. Called automatically
     *  by Struts framework.
    */
    public ActionErrors validate(ActionMapping mapping,
                                 HttpServletRequest request){

       //create a blank ActionErrors
       ActionErrors errors = new ActionErrors();

       //check for a blank user id
       if( null == _userid ){
          errors.add("userid",
                     new ActionMessage("reg.error.userid.missing"));
       }

       //check password 1 == password 2
       if( !_pwd.equals(_pwd2)){
          errors.add("password",
                     new ActionMessage("reg.error.password.mismatch"));
       }

       return errors;
    }
}
```

Recall that the Registration webapp's user input consisted of three fields: a user ID, a password, and the password confirmation. The first thing to note from Listing 6-1 is that this `ActionForm` subclass contains getters and setters for each of these three fields.

The body of `validate()` follows a simple pattern:

1. Create a blank `ActionErrors` instance.
2. Run a check for each field, and populate the `ActionErrors` instance with error messages.
3. Return the `ActionErrors` instance.

In Listing 6-1, I've only put in a check for a blank user ID and a test to see if the password and its confirmation were the same. In a real application, you might want to incorporate further checks to ensure that the password isn't too short or too long and that the user ID doesn't contain nonalphanumeric characters. As you can see, even for a trivial application like this, you'd need quite a few checks!

One thing you *should not* put in here is the check to see if the user ID is taken. You *could* actually do this, for example, by adding the code shown in Listing 6-2.

Listing 6-2. *Code to Check for a Duplicate User ID*

```
//test for duplicate userid
if(User.exists(_userid)){
   errors.add("userid",
           new ActionMessage("reg.error.userid.exists"));
}
```

But you *should not* do this because it isn't a simple validation, since it involves calling Model code (the `User` class from Listing 5-1):

```
if(User.exists(_userid)){...
```

and therefore, should be excluded from your subclass of `ActionForm`. This rule (you could call it a "best practice") has practical benefits. You'll see this later when I describe the Validator framework in Part 2 of this book.

To summarize, each form displayed to the user is associated with an `ActionForm` subclass. This subclass stores the form data, and exposes a `validate()` function that Struts calls automatically in order to run validations.

There's still one loose string, which is `ActionErrors`, and I'll describe this next.

Using ActionErrors

Struts uses the `ActionErrors` instance returned by `validate()` to display the error messages on the redisplayed form.

`ActionErrors` (note the plural) is very much like a `HashMap`, and it is used to store error messages, indexed by error keys. The error message is not a `String`, but an instance of `ActionMessage`.

■ **Note** In earlier versions of Struts, the error message was an instance of `ActionError` (note the singular), but that's been deprecated in favor of `ActionMessage`.

`ActionMessages` are added to an `ActionErrors` instance using the `add()` function:

```
public void add(String key, ActionMessage message)
```

You can think of this as analogous to the `put()` function on a `HashMap`.

The constructor for `ActionMessage` (see Listing 6-1) might seem a little cryptic:

```
new ActionMessage("reg.error.password.mismatch")
```

For one thing, it's obvious that we're not passing in the actual error message! Instead, we're passing in a *key* to an error message, stored in a *properties file* accessible to Struts.

■ **Note** Exactly how Struts locates this properties file is the subject of Chapter 9, where you'll learn how to configure Struts.

Properties files are just text files, which contain key/value pairs. In this case, the *key* is `reg.error.password.mismatch`, and the *value* might be `The passwords you keyed in don't match!`.

Listing 6-3 shows a possible properties file for the Registration webapp.

Listing 6-3. *registration.properties, a Properties File for the Registration Webapp*

```
# Properties file for Registration webapp


# Error messages:

reg.error.userid.missing=The user id is missing.
reg.error.userid.exists = The user id exists. Choose another.
reg.error.userid.bad = Use only alphanumerics for the userid.

reg.error.password.mismatch = The passwords you keyed in don't match!
reg.error.password.long = The password is too long!
reg.error.password.short = The password is too short!

# Prompts:

reg.prompt.userid=User ID
reg.prompt.password=Password
reg.prompt.password.confirmation=Password Confirmation
```

In Listing 6-3, you can see how error message keys are associated with the actual error messages. You can also see how prompts that go into the user interface can be stored in the properties file. Essentially, using properties files is a simple, general-purpose way to store static key/value pairs for any Java application.

■ **Note** The dotted notation for the property keys (e.g., `reg.error.userid.bad`) is just a convention. This convention is useful because it forms a namespace for your properties files. If you had two or more properties files, following this convention would prevent you (or your development team) from using the same key twice. The convention is also useful because it gives you an idea of what the key might refer to.

The string argument of the `ActionMessage` constructor *must* be a key on a properties file. This might seem restrictive, but there's a very good reason for this. This is how Struts makes it easy to localize an application.

For example, if you wanted a German version of the Registration webapp, all you'd have to do would be hand over this properties file (`registration.properties`) for translation. You'd save the translated version as `registration_de.properties`, and bundle it with your webapp. German users would see the German version of the Registration webapp and everyone else would see the default English version. Localizing a webapp was never so easy! We'll see this in action further down the road in Chapter 12.

■ **Note** The use of properties files also makes your webapp more consistent, since it lessens your chances of displaying "Your user ID is bad" in one error message and "Your user ID isn't valid" for the same error in a different form.

But `ActionMessages` are only one half of the puzzle of how error messages are displayed.

The other crucial bit is the **key** under which the `ActionMessage` is stored on the `ActionErrors` instance. With reference to Listing 6-1:

```
errors.add("password", new ActionMessage(...));
```

As you might guess, the key corresponds to the name of the form property associated with the error. This is only a helpful convention, not a necessity. Struts uses a special Struts tag called `<errors>` with an attribute called `property` that you match with the error key, like so:

```
<html:errors property="password"/>
```

You'd place this tag on the JSP containing the form, near the password field. I'll go into this in more detail in Chapter 8, when I describe the View components of Struts.

Figure 6-2 summarizes the relationship between `ActionErrors`, `ActionMessage`, properties files, error keys, and the error tag.

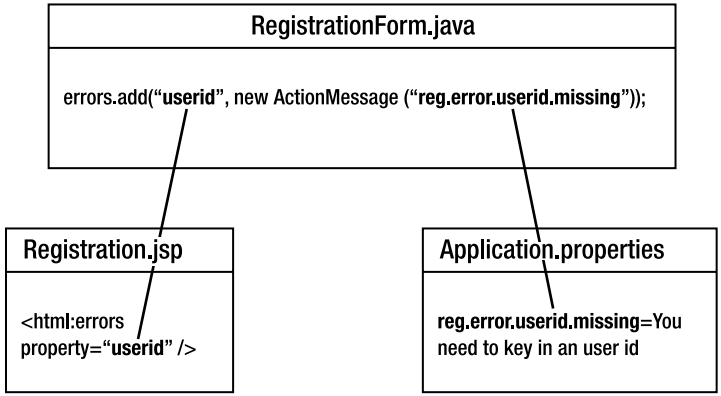

Figure 6-2. *How ActionErrors ties error messages to the View component*

To summarize, when the user submits a form, Struts calls `validate()` on the associated `ActionForm` subclass, which returns an `ActionErrors` instance. This is just a collection of `ActionMessages`, each of which references error messages stored in properties files. When Struts redisplays the page, it uses special `<errors>` tags you've embedded on the page to locate exactly where to display the error messages. The `<errors>` tags on the page are distinguished by their `property` attribute, which forms the link between the tag and a particular `ActionMessage` instance stored on the `ActionErrors` collection.

That's quite a lot to take in one go! You might want to try the quiz that follows to see if you've really understood the preceding material.

Simple Validation Quiz

Q1: Suppose validating a certain field requires a numerical calculation. Would this qualify as a simple validation? You should give clear reasons for your answer.

Q2: When is `validate()` called? Before Struts populates the `ActionForm` or after?

Q3: If `validate()` returns an empty `ActionErrors` instance, will the form be redisplayed?

Q4: Rewrite Listing 6-1 so that Struts only displays the error message for the *first* validation error encountered.

Q5: If you changed the name of one of a form's properties (say `zipcode` to `postcode`), would you have to change the corresponding error key for `ActionErrors`, that is, `errors.add("zipcode",...)` to `errors.add("postcode",...)`?

Q6: If your answer to Q5 was yes, what other changes would you have to make?

Q7: If there were more than one validation error for a single property, which error message (if any) do you think would be displayed?

Check your answers with those in Appendix D. Once you're confident of the material, try out Lab 6.

Lab 6: ContactForm for LILLDEP

LILLDEP (Little Data Entry Program) is a webapp that maintains a database of *contacts*. Each contact has the following 14 properties:

- `name` of contact
- `designation` of contact (e.g., "Senior Engineer")
- `department`
- `email` address
- `tel` number
- `fax` number
- `company` to which contact belongs to
- `address` of contact's company
- `postcode` of company address ("zip code" in the United States)
- `country` of company site
- `website` of contact's company
- `activity` engaged in by contact's company
- `classification` of contact's company
- `memo`, which contains miscellaneous notes for this contact

Figure 6-3 shows the form data as it is seen by a user.

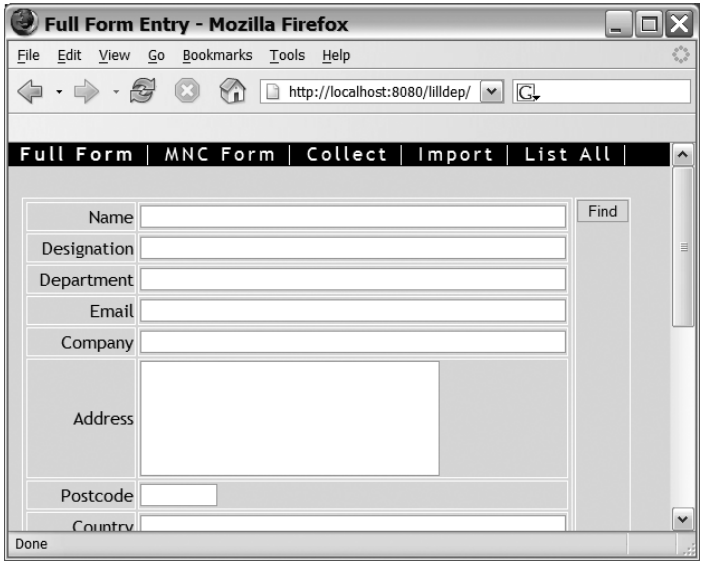

Figure 6-3. *The LILLDEP start page*

The associated `ActionForm` subclass that handles validation for this contact data is called `ContactForm`. In this lab, you'll complete the implementation for `ContactForm`. There are three things you will need to do:

- Put in the `getXXX` and `setXXX` functions corresponding to each of the 14 properties.
- Complete the `validate()` function.
- Complete the implementation of a `reset()` function, which resets the values of the `ContactForm`.

Complete this lab by following these steps.

Step 1: Prepare the Development Environment and Scripts

1. Copy the `lilldep.zip` file from the Source Code section of the Apress website, found at `http://www.apress.com`, into the Struts directory you created in Lab 4.
2. Unzip the contents, making sure you've preserved the directory structure.
3. You should see a subdirectory called `.\Struts\lilldep\`.
4. Open `.\Struts\lilldep\compile.bat` in a text editor and amend the `PATH` environment variable so that it points to your JDK installation.

■ **Note** On Windows XP, ME, or 2000, which have built-in unzipping capabilities, the zip file may add an extra `lilldep` directory in the unzipped path. So, the `compile.bat` file's path would be `.\Struts\lilldep\lilldep\compile.bat`. You could move the `lilldep` folder up or just continue. The compile scripts should work regardless.

Test your changes by clicking on `compile.bat`. You should get no errors, and you should see a file called `lilldep.war` in `.\Struts\lilldep\`.

In what follows, I'll refer to all paths relative to `.\Struts\lilldep\`.

Step 2: Implement Getters and Setters for ContactForm

A skeleton for `ContactForm` has been created for you, in the directory `net\thinksquared\lilldep\struts`.

Now, *unlike* the example with the Registration webapp (Listing 6-1) where the form data was stored in private variables such as `_userid` and `_pwd`, the getters and setters of `ContactForm` must store data in a bean called `Contact`.

■ **Note** The `Contact` bean was autogenerated using Lisptorq. If you'd like details on how it and other Model classes were created, refer to Appendix A.

The `Contact` bean also contains getters and setters for each of the 14 properties listed earlier. Your getters/setters for `ContactForm` should call the corresponding getters/setters for the `Contact` bean contained in `ContactForm`. This may seem a little strange, but this simplifies the code in later lab sessions.

Complete the following steps:

1. Put in the getters and setters for `ContactForm`.
2. Put in an extra pair of functions, `getContact()` and `setContact()`, to get and set the `Contact` bean instance.
3. Run `compile.bat` to ensure you have no errors.

Step 3: Implement validate()

You need to implement a few validations:

1. Ensure that the company name isn't blank. The error key is `lilldep.error.company`.

2. Ensure that the contact's name isn't blank. The error key is `lilldep.error.name`.

3. Ensure that the email address, **if** not blank, is a valid email address. The error key is `lilldep.error.email`.

Check that your work compiles without errors.

These are only a few possible validations you might in reality run on `ContactForm`. In Chapter 15, you'll see how to write simple validations *without* having to use Java.

Step 4: Implement reset()

In addition to `validate()`, `ActionForm` exposes another function, called `reset()`, which resets the `ActionForm`'s values to defaults you define. `reset()` can be called by View code using a Reset button.

The skeleton for this function is in `ContactForm`. Complete the implementation to reset the properties of the `Contact` bean. (Hint: You should peruse the source code for `Contact` to find a suitable function that does this.)

As usual, compile your code to ensure that it's free of errors.

What Has Struts Done for You Today?

Just as in life, "counting your blessings" often puts your problems in perspective, so it may be helpful to consider what Struts has done thus far to make writing web applications easier:

- **Automatically transfers form data into your app**: When the user submits a form, the data is automatically read from the form and put into your `ActionForm` subclass. Struts even performs type castings for you.

- **Centralizes simple validations**: All your simple validations go into the `validate()` function. It can't get any simpler. Well, it can, as you'll see in Chapter 15.

- **Automatically redisplays badly filled forms with errors messages**: Without Struts, you had to do this yourself. Consider for a few moments how you might do this yourself without Struts. I hope you can see this is a very big plus!

- **Enables easy localization of error messages**: This occurs through the use of properties files.

Summary

- `ActionForm` is the central class that for simple validation.
- Your `ActionForm` subclass must have getters and setters for each form property, and must override `validate()` to run simple validations.
- `ActionErrors` is the return value of `validate()` and holds error messages.
- Individual error messages are represented by instances of `ActionMessage`.
- The actual error message is stored in properties files.

CHAPTER 7

Processing Business Logic

In the previous chapter, you learned how to run simple validations on user input. Within the Struts framework (or any sensible framework, really), only after all simple validations pass is the data ready to be processed by the system. This processing, usually referred to as "business logic," consists of three broad tasks:

- **Complex validations**: These are further validations that need to be run but, unlike simple validations, are not generic. They require domain-specific processing, or communication with the Model components of your webapp.
- **Data transformations**: This includes making calculations, saving data, and preparing data for output. This is the meat of a particular task.
- **Navigation**: This involves deciding what page to display next to the user at the completion of the task.

Struts centralizes these three tasks in a single Controller class called `Action`.

1,2,3 Action!

Just as simple validation was performed by your subclass of `ActionForm`, the processing of business logic is done by your subclass of

`org.apache.struts.action.Action.`

Within the MVC paradigm, your `Action` subclass is a Controller (see Chapter 5, Figure 5-5), because it communicates with both the Model and View components of your webapp through well-defined interfaces.

The input data upon which your `Action` subclass acts is the form data submitted by the user. As you'll recall from the previous chapter, form data submitted by the user is automatically transferred by Struts to your `ActionForm` subclass. This `ActionForm` subclass instance contains the input data for your subclass of `Action`. The `ActionForm` subclass is a JavaBean, meaning it has `getXXX()` and `setXXX()` functions corresponding to each property

on the HTML form. Your `Action` subclass can use these `getXXX()` functions to read the form data. Figure 7-1 illustrates this data flow.

Register New User - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/registratior
Register New User
You are successfully registered with the userid anteater
Done

Figure 7-1. *Data flow from form to Action*

In fact, for each `Action` subclass you define, you may associate any number of `ActionForm` subclasses, all of which are potential inputs into that `Action` subclass. You'll see how this association is made in Chapter 9.

The Statelessness of Action

One very important thing to know about `Action` is that it must be stateless. You must never store data in your `Action` subclass. In other words, your `Action` subclass must never have *instance variables* (variables within a function are OK). Listing 7-1 illustrates what you should *never* do.

Listing 7-1. *Never Use Instance Variables in an Action Subclass!*

```
public class MyBadAction extends Action{

   private String myBadVariable; // NEVER EVER do this!

   protected void myFunction(){
      String s = null; //variables in a function are OK.
      ...
```

```
    }

    ...//rest of Action
}
```

Struts manages the creation of your `Action` subclasses, and these are pooled (reused) to efficiently service user requests. For this reason, you can't use instance variables. There is no way to guarantee that the data you store in an instance variable applies to a given request.

Subclassing Action

There is just one method you need to override in your `Action` subclass, and that's `execute()` (see Listing 7-2).

Listing 7-2. *The execute() Function*

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
```

■ **Note** In older versions of Struts (version 1.0 and earlier), `execute()` was called `perform()`.

After the `validate()` method on the `ActionForm` passes, Struts calls `execute()` on the `Action` associated with the form. As you can see, five classes play a role in this function:

- `ActionForward`: Represents the "next" page to be displayed. This is the return value of `execute()`.
- `ActionMapping`: Represents associations between this `Action` and its possible "next" pages.
- `ActionForm`: Contains input form data.
- `HttpServletRequest`: Contains request-scoped and session-scoped data. You can also place data on the session using this class. Note that not all data goes on the submitted form. Other data could be placed on the form's "action" URL. You can read these with `HttpServletRequest`.

- `HttpServletResponse`: Allows you to write data to the user's web browser. You are less likely to use this class, unless you want to send, say, a generated PDF report back to the user. ()

The important functions of each of these classes are listed in Appendix B.

Business Logic in the Registration Webapp

We'll now take up our earlier example of the Registration webapp (see Chapters 5 and 6), and see how we can process business logic. For the Registration webapp, this involves the following:

- **Complex validation**: Checking if the given user ID exists. I gave you a solution in Chapter 6, Listing 6-2, which we'll use here.
- **Data transformation**: Saving the user ID and password into the database, using the `User` data object (see Listing 5-1 in Chapter 5).
- **Navigation**: Displaying the "You're Registered!" page (Figure 7-2) if registration succeeds, or redisplaying the form with an error message (Figure 7-3) if the user ID exists.

Register New User - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/registratior
Register New User
You are successfully registered with the userid anteater
Done

Figure 7-2. *The "You're Registered!" page*

Figure 7-3. *Redisplay of form to user with error message*

We only need one `Action` subclass, called `RegistrationAction`, which is shown in Listing 7-3.

Listing 7-3. *RegistrationAction.java*

```
package net.thinksquared.registration.struts;

import javax.servlet.http.*;
import org.apache.struts.action.*;
import net.thinksquared.registration.data.User;

public class RegistrationAction extends Action{

   public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response){

      //get userid and password
      RegistrationForm rForm = (RegistrationForm) form;
      String userid = rForm.getUserId();
      String password = rForm.getPassword();
```

```
        //complex validation: check if userid exists
        if(User.exists(userid)){
           ActionMessages errors = new ActionMessages();
           errors.add("userid",
                      new ActionMessage("reg.error.userid.exists"));

           saveErrors(request,errors);

           //navigation: redisplay the user form.

           return mapping.getInputForward();

        }else{

           //data transformation: save the userid and password
           //to database:

           User user = new User();
           user.setUserId(userid);
           user.setPassword(password);
           user.save();

           //navigation: display "you're registered" page

           return mapping.findForward("success");
        }
     }
}
```

Take your time studying Listing 7-3, because it is a pattern for every `Action` you'll ever write.

The first thing to note is the cast from `ActionForm` to `RegistrationForm`:

```
RegistrationForm rForm = (RegistrationForm) form;
```

Recall that the `ActionForm` passed into `execute()` is really a `RegistrationForm` instance (see Listing 6-1), and that it contains form data that has passed simple validation. The cast is done so that the `userid` and `password` properties of the form can be read.

■ **Note** This isn't the best way to read data from an ActionForm. In Chapter 17, I'll describe a better alternative.

I haven't yet shown you how Struts knows that the form data goes into and is validated by `RegistrationForm`. And I also haven't shown you how Struts knows that the populated `RegistrationForm` needs to be sent to `RegistrationAction` for further processing. You'll see how both these mappings are made in Chapter 9.

I'll analyze the rest of Listing 7-3 using the three broad tasks every `Action` subclass performs: complex validation, data transformation, and navigation.

Complex Validation

The relevant section performing complex validation in Listing 7-3 is shown in Listing 7-4.

Listing 7-4. *Complex Validation in RegistrationAction*

```
if(User.exists(userid)){
         ActionMessages errors = new ActionMessages();
         errors.add("userid",
                    new ActionMessage("reg.error.userid.exists"));

         saveErrors(request,errors);

         //navigation: redisplay the user form.

         return mapping.getInputForward();

      }else{ ...
```

The `User.exists()` function checks if the user ID given exists. This is typical of what I mean by "well-defined" interfaces between the Controller (`RegistrationAction`) and Model (`User` class).

Struts does not prevent you from using raw SQL to check if a user ID exists, but this would be a poorer implementation of the MVC design pattern. Having a Model class handle data access like this makes your code cleaner, much more maintainable, and more future-proof. If you switched to a database that used a different variant of SQL, you would need to change at most the implementation of the `exists()` function and not every single SQL statement checking for a duplicate user ID.

The next statement creates a blank list to hold error messages:

```
ActionMessages errors = new ActionMessages();
```

The use of `ActionMessages` to hold error messages instead of `ActionErrors` is rather unfortunate. In earlier versions of Struts, it would have indeed been `ActionErrors`, but later versions have deprecated the use of `ActionErrors` everywhere except as the return value of `validate()` in `ActionForm`.

After creating the holder for error messages, I next add an error message to it:

```
errors.add("userid", new ActionMessage("reg.error.userid.exists"));
```

This should be old hat to you by now. If you have difficulty understanding this statement, you should reread the "Using ActionErrors" section in Chapter 6.

The last statement I'll consider is

```
saveErrors(request,errors);
```

You signal a complex validation failure by using `saveErrors()` to save a list of errors. However, unlike simple validation where the form is automatically redisplayed, here the list of errors is used to populate the `<html:errors>` tags on the "next" page.

■ **Note** `saveErrors()` actually saves the errors onto the request object, which is an instance of `HttpServletRequest` that was passed as a parameter into `execute()`.

After `execute()` completes, Struts examines the request object for errors, then tries to display them in the "next" page indicated by the `ActionForward` instance returned by `execute()`.

Struts knows *which* error messages to display and *where* to display them because the "next" page is assumed to contain our old friend, the `<html:errors>` tags. We'll revisit this in detail the next chapter. If there are no `<html:errors>` tags, then no error messages are displayed.

The astute reader may be wondering why it's `saveErrors(request, errors)` instead of simply `saveErrors(errors)`. The answer is that `Action` subclasses must be stateless. The errors object must therefore be stored on request.

Data Transformation

In the `RegistrationAction` class, the only data transformation involved is saving the user ID and password to the database, as Listing 7-5 illustrates.

Listing 7-5. *Data Transformation in RegistrationAction*

```
User user = new User();
         user.setUserId(userid);
         user.setPassword(password);
         user.save();
```

As you can see, using Model classes to write data results in clean code. Model classes are not part of Struts, because this is a task best left to dedicated persistence frameworks like Hibernate or Torque. You should refer to Appendix A for details.

Navigation

Navigation refers to programming logic deciding what page to display next to the user. In `RegistrationAction`, there are just two possible "next" pages: the page containing the submitted form (redisplayed with error messages) and the "You're Registered!" page indicating successful registration.

The code used to reach each is slightly different. Listing 7-6 shows the navigation logic involved.

Listing 7-6. *Navigation in RegistrationAction*

```
if(User.exists(userid)){

        ...

        //navigation: redisplay the user form.
        return mapping.getInputForward();
}else{

        ...

        //navigation: display "You're registered" page

        return mapping.findForward("success");

}
```

`ActionForward`, the return value of `execute()`, is a reference to the "next" page. Illustrated in Listing 7-6 are the two most important ways of instantiating it.

■ **Note** If you look up the JavaDocs for `ActionForward`, you'll find that there are in fact a number of different constructors for `ActionForward`. Each constructor allows you to change defaults such as whether or not the page is a redirect. There are even subclasses of `ActionForward` that create `ActionForwards` with the more common choices.

The first involves creating an `ActionForward` reference to the page containing the submitted form. Possibly the only reason you'd want to do this is to redisplay a form if it fails complex validation:

```
mapping.getInputForward()
```

To get a reference to the input page you use the `getInputForward()` function on the `ActionMapping` instance, `mapping`. Recall that `mapping` is supplied for you by Struts, because it's a parameter in `execute()`.

Notice that unlike simple validation where redisplay was automatically handled by Struts, here you're on "manual," meaning you have to specify the error page yourself. This is because of the very different natures of simple and complex validation. For example, you might check a user's account balance before performing a task. If there are insufficient funds, it makes no sense to automatically redisplay the input page. A more appropriate response would be to display an "insufficient funds" page.

The second way you'd want to instantiate `ActionForward` is for it to point to a *named* "next" page:

```
mapping.findForward("success")
```

Here, the name `success` refers to a named page, which is one possible "next" page I've bound to `RegistrationAction`. I'll describe how this is done in Chapter 9.

This indirect way of referring to a page might seem overly complicated. Why not just use a direct reference like

```
new ActionForward("/mywebapp/registered.jsp")
```

There are two good reasons why you should avoid using a direct reference. The first is because the indirect method makes your webapp more maintainable, and the second is that it promotes code reuse.

To understand the first reason, you'd have to know that the named pages are specified in one file called `struts-config.xml`. So, if you moved a page, you'd only have to amend this one file. If the paths were hardwired, you'd have to change several `Action`s and recompile your app! Not good.

The second reason is more subtle, but more compelling. It is often the case that you want to reuse an `Action` in different contexts. Depending on the context, you would want a different "success" page to be displayed. Hardwiring a "next" page into an `Action` obviates the possibility of doing this.

To recap, there are two important ways to display the "next" page:

- **The input page**: `mapping.getInputForward()`
- **A named page**: `mapping.findForward(...)`

Lab 7: Implementing ContactAction for LILLDEP

This lab continues where we left off in Lab 6. `ContactAction` is the `Action` subclass associated with `ContactForm` of Lab 6.

In this lab, you will complete the implementation of `ContactAction`. Before you begin, ensure that you can compile the LILLDEP webapp without errors. Then complete the following:

1. Get the `Contact` data object from the `ContactForm` and save it. You might want to peruse the source code for `BaseContact` to see how to save a `Contact`.
2. If an `Exception` is thrown while saving a `Contact`, redisplay the input page with the error message key `lilldep.error.save`, for the property `ActionMessages.GLOBAL_MESSAGE`.
3. If all goes well, forward to a page named "success."
4. Run `compile.bat` to ensure your work compiles with no errors.

Summary

In this chapter, you've learned how business logic is processed by your `Action` subclass. The highlights of this chapter are as follows:

- `org.apache.struts.action.Action` subclasses must be stateless.
- You need to override just the `execute()` method of `Action`.
- Complex validation errors are flagged using the `saveErrors()` function.
- Redisplay of a page with errors is "manual."
- Use `mapping.getInputForward()` to redisplay the input page.
- Use `mapping.findForward(...)` to display a named page.

CHAPTER 8

Basic Struts Tags

Struts implements the View component of the MVC design pattern entirely through the use of custom tags (actually, the correct term is custom *action*, but I think the term custom *tag* conveys more meaning, and leaves less room for confusion with Struts `Actions`). These tags are applied to the JSPs that constitute the View component of your webapp. The Struts tags are bundled into *five* tag libraries:

- **HTML**: The custom tags in the HTML library are essentially in one-to-one relationship to the ordinary HTML `<form>` tag and its associated input tags, like the various `<input>` tags. The purpose of this tag library is to enable you to connect your View component to the Controller components described in Chapters 6 and 7. The actual connecting details are described in Chapter 9.

- **Bean**: This library has custom tags primarily for writing text. There are two reasons why you'd use the Bean tags instead of hard-coding text into your JSPs. The first is to enable internationalization, that is, the display of the View component in multiple languages. We'll show how Struts helps you internationalize your webapps in Chapter 12. The second reason is to avoid using scriptlets to display the contents of objects stored on the request or session objects.

- **Logic**: This library provides tags for conditional processing and looping. You use tags in the Logic library in lieu of using scriptlets. The logic library tags are much easier to use and result in significantly more readable code.

- **Nested**: This library has tags for displaying "nested" properties of a form or object. We'll describe this in detail in Chapter 10.

- **Tiles**: This library contains tags that allow you to create layouts. You'll see how to use the Tiles tag library in Chapter 14.

■ **Note** Appendix C is a comprehensive reference for all Struts tags.

In this chapter, we'll only discuss the most frequently used tags from just the HTML and Bean libraries.

Before we go into the details of each of these tags, you first have to understand how a JSP page is processed by Struts.

Page Processing Lifecycle

When Struts is asked for a page, it first replaces all Struts tags on that page with the necessary textual data. The underlying mechanism of this replacement process was described in Chapter 4. Not all Struts tags will show up on the displayed page. Some have preconditions, and will only display if they are satisfied. For example, the `<html:errors>` tag displays only if there are error messages to be displayed.

For a page that doesn't contain a form, this tag replacement stage ends that page's processing lifecycle.

If a page *does* contain a form, then Struts handles processing of the form data. This happens in essentially two stages, which have been the subject of the previous two chapters: simple validation and the processing of business logic.

The form data is first run through simple validation (on your `ActionForm` subclass). If this fails, the page is *typically* redisplayed automatically.

■ **Note** Whether simple validations are indeed performed and which, if any, input page is redisplayed are all controlled by the Struts configuration file, to be described in Chapter 9.

Because the page now contains validation errors, any relevant `<html:errors>` tags on the page will display their error messages.

Once the user corrects and resubmits the form, the data is again validated. As before, if the form still contains errors the page is redisplayed yet again. This cycle of redisplaying the form and validating it continues until the form data is free of simple validation errors.

When this happens, Struts sends the form data for business logic processing (by your `Action` subclass), and the "next" page is displayed. Note that the "next" page could be an error page or could have error messages in the event of complex validation errors.

Figure 8-1 summarizes these steps.

To make these ideas more concrete, we'll continue with the Registration webapp example of the previous chapters. Along the way, we'll show you how to use the most basic Struts tags from the HTML and Bean libraries. Again, Appendix C is a comprehensive reference for all Struts tags.

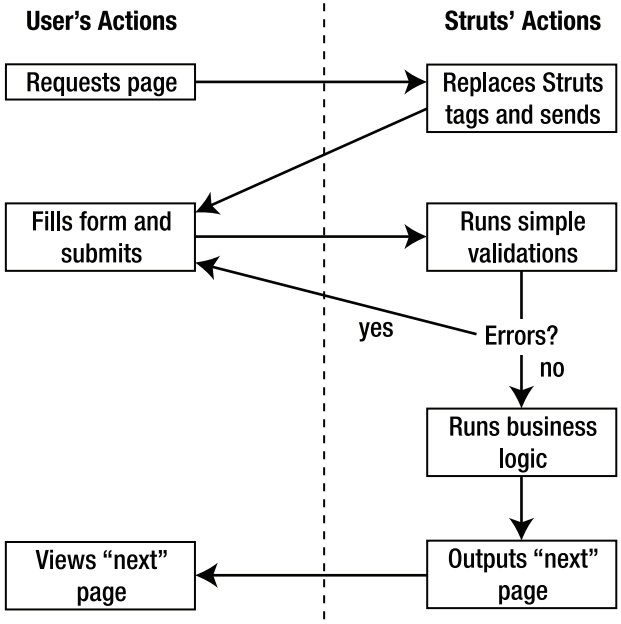

Figure 8-1. *Typical Struts page processing lifecycle*

Evaluation, Replacement, and Sending

When a Struts JSP page is requested, its processing does *not* involve the main Struts servlet (`ActionServlet`). Only submissions of form data involve the main servlet. It is this servlet that populates your `ActionForm` subclass and invokes `execute()` on your `Action` subclass.

Of course, Struts classes are involved in a page's rendering, but these are just the `BodyTagSupport` and `TagSupport` classes representing each tag (see Chapter 4). The only state that these classes know about is stored on the server-side `HttpServletRequest` and `HttpSession` objects, representing the current request and session, respectively.

As you've seen in Chapter 4, `BodyTagSupport` or `TagSupport` are responsible for the final rendering of a tag's visual appearance. Not all Struts tags have a visual appearance, though. Those from the Logic library (discussed in Chapter 10), for example, perform conditional processing to selectively display other tags that *do* have a visual appearance.

The processing of a Struts JSP page may be summarized in three steps:

- **Evaluation**: Some Struts tags (from the Logic library) or the `<html:errors>` tag evaluate themselves to determine if they should indeed be displayed.
- **Replacement**: Data is read from the request or session objects (or global forwards—more on this in Chapter 9) and is pasted into the rendered page. `<html:errors>`, for example, pastes in the appropriate error message.
- **Sending**: The final page is sent to the user.

These three steps are always involved when a Struts JSP page is requested.

The View Component of the Registration Webapp

The View component of the Registration webapp (see Chapter 5) consists of a single page, `registration.jsp`, which contains a form with three inputs: a textual input for a user ID and two password fields.

The JSP code for this is given in Listing 8-1, and the visual output, as seen by the user, is depicted in Figure 8-2. Take some time to study both before reading further.

Listing 8-1. *Registration.jsp*

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html>
<head>
   <title><bean:message key="registration.jsp.title"/></title>
</head>
<body>
      <h1><bean:message key="registration.jsp.heading"/></h1>
      <html:form action="Registration.do" focus="userid">
         <p>
           <bean:message key="registration.jsp.prompt.userid"/>
           <html:text property="userid" size="20" />
           <html:errors property="userid" />
         </p><p>
           <bean:message key="registration.jsp.prompt.password"/>
           <html:password property="password" size="20" />
           <html:errors property="password" />
```

```
        </p><p>
          <bean:message key="registration.jsp.prompt.password2"/>
          <html:password property="password2" size="20" />
        </p>

        <html:submit>
           <bean:message key="registration.jsp.prompt.submit"/>
        </html:submit>

        <html:reset>
           <bean:message key="registration.jsp.prompt.reset"/>
        </html:reset>
      </html:form>
</body>
</html:html>
```

Figure 8-2. *registration.jsp as it appears to the user*

Declaring and Installing the HTML and Bean Libraries

Listing 8-1 starts with defining the prefixes for the Bean and HTML tag libraries:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
```

Having written your own custom tags in Lab 4, this should hold no surprises for you.

The TLD files and the Java handlers are all bundled in your Struts distribution:

- The TLD files should be placed in `\WEB-INF\`.
- Struts comes with a `web.xml` file that you should use. At the time of this writing, this `web.xml` file contains the appropriate `<taglib>` sections declared for your convenience. But you should always check this for yourself, and insert `<taglib>` sections for the five tag libraries if necessary.
- The Java handler classes for these tags are in the various JAR files that comprise the Struts binaries, which you should place in your webapp's `\lib\` directory (see Chapter 2).

■ **Caution** Do *not* place the Struts JAR files in your servlet container's "shared" or "common" directories. They must *not* be shared. You must deploy a fresh set in the `\lib\` directory of each webapp. Sharing might in some cases cause `ClassNotFoundException`s to be thrown by the servlet container.

Displaying Static Text

The first thing you might notice in Listing 8-1 is that the prompts in Figure 8-2 (like Register New User, etc.) do not appear in the listing. Instead, `registration.jsp` uses `<bean:message>` tags to display these. For example, the tag

```
<bean:message key="registration.jsp.prompt.userid"/>
```

displays as

```
Please enter a userid (between 5 to 15 characters)
```

The `<bean:message>` tag can be used anywhere to display static text. The `key` attribute on this tag points to a key-value pair on a resource file `Application.properties`, just as it did in the case of error messages of Chapters 6 and 7.

Of course, there's nothing to stop you from hard-coding text in your JSPs, but this would make your app much more difficult to internationalize and maintain.

■ **Note** Struts' simple approach to displaying static text will only you take you so far. If you have *lots* of static text to display, then using a template engine like FreeMarker or Velocity (both of which I will *not* cover in this book) will be necessary. Refer to the "Useful Links" section for some web resources.

The Bean tag library contains other tags besides `<bean:message>`. Probably the only other tag you'd use is `<bean:write>`, which allows you to display properties of JavaBean objects. Refer to Appendix C for a description of this tag.

Forms and Form Handlers

Webapps are all about getting user input and processing them. The primary way of doing this with Struts (as with HTML) is through a *form*.

In Struts, forms are declared within an `<html:form>` container tag. This is similar to the way you'd declare a form in HTML. The `<html:form>` element has the `action` attribute that describes the handler that processes the form data:

```
<html:form action="Registration.do" ...
```

In this snippet, the handler is named `Registration`. A handler is a *particular combination* of your `ActionForm` subclass that performs simple validation and your subclass of `Action` that does business logic. You will see how to declare a handler in the next chapter. For now, you only need to understand that the `action` attribute points to a handler that you've defined.

Handlers traditionally end with the `.do` prefix, which tells the servlet container that the `action` is to be handled by Struts. This mapping is done in `web.xml`, and is shown in Listing 8-2 (which also shows the Struts `ActionServlet` declaration). These declarations are done for you in the `web.xml` file that comes with Struts. However, a high-level understanding of the details is useful, so I'll digress a little to do this.

■ **Note** In case it isn't clear, Listing 8-2 is an excerpt from the web.xml file generously made available by the Apache Software Foundation. The Apache License is available at `http://www.apache.org/licenses/LICENSE-2.0`.

Listing 8-2. *Struts' Standard servlet and servlet mapping declarations*

```
<!-- Standard Action Servlet Configuration  -->
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
      org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
```

```
<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Listing 8-2's `<servlet>` declaration starts the main Struts servlet class (`org.apache.struts.action.ActionServlet`) and gives it a reference to the Struts configuration file (`struts-config.xml`) described later in Chapter 9. This `<servlet>` declaration is given the name action. The following `<servlet-mapping>` tag refers to this name, and specifies that any incoming URL with the page extension ending with `.do` is to be referred to the servlet called `action` for processing.

Remember that it's the servlet container (Tomcat, in our case) that reads the `web.xml` file and performs these instructions. The servlet container is also the one that orchestrates the control flow between the user and Struts, as depicted in Figure 8-1. It knows how to do this from the `<servlet>` and `<servlet-mapping>` declarations in Listing 8-2.

The last thing to note concerning the `<html:form>` tag is the attribute `focus="userid"`. This is a directive telling Struts to *autogenerate* JavaScript to cause the `userid` field on the form to grab focus (that is, the cursor appears in this field) when the page loads in the user's web browser. The details of the generated JavaScript aren't important, but the fact that Struts does this at all *is*. As you will see in this book, much of the power of Struts is in little conveniences like this.

Data Input Tags

Struts tags that accept data input must be placed within the `<html:form>` tag. Listing 8-1 contains four input tags:

- `<html:text>` represents a text field.
- `<html:password>` represents a password field. The text in this field is displayed as asterisks.
- `<html:submit>` represents the submit button.
- `<html:reset>` represents a reset button to clear the form.

There are a number of other input tags (e.g., for radio buttons, lists, and drop-down lists), which are described in detail in Appendix C and in the lab sessions of subsequent chapters. In this chapter, we'll concentrate on these four input tags only.

The textual field tags `<html:text>` and `<html:password>` all have a `property` attribute that ties them to the field on the form. Each field *must* correspond to a `getXXX()` and `setXXX()` pair on the `ActionForm` subclass associated with the form handler. Exactly *how*

the form handler links the JSP page with an ActionForm subclass is the subject of Chapter 9. For now, simply assume that each <html:form> has input fields corresponding to properties of an ActionForm subclass.

For example, from Listing 8-1, we can deduce that the ActionForm subclass that handles the Registration data must have the functions getUserid() and setUserid(). And it does (refer to Listing 6-1).

The text fields also accept a size attribute to allow you to specify the physical extent of the displayed field. This is just as you would expect for a text input field in HTML. In fact, all the input tags have attributes closely matching their HTML counterparts.

The <html:submit> and <html:reset> tags represent the submit and reset buttons on the form, respectively. The <bean:message> tags enclosed by these tags tells Struts what labels to place on the buttons. Again, this approach allows easy internationalization, and also lends a more uniform look to your webapp.

The reset button calls the reset() function on your ActionForm subclass (see step 4 of Lab 6). If one isn't defined, the super.reset() function is called, which simply redisplays a blank form.

Displaying Errors

In the previous two chapters, you saw how validation error messages are generated. We'll now describe how they are displayed.

When a form fails validation (either simple or complex), an ActionErrors or ActionMessages object is created containing the error messages. Both ActionErrors and ActionMessages behave like Java HashMaps. The key of the HashMap corresponds to the property attribute of the <html:errors> tag. The associated value is the error message itself.

However, unlike a HashMap, it's possible to store *more* than one error message under the same key. This is obviously useful if there's more than one error associated with a single field.

When Struts loads a page, it checks if there are ActionErrors or ActionMessages for the page. If there are, it dutifully pastes the error messages in the right <html:errors> tags of the page. Figure 8-3 shows this process in action.

As you can see, only errors whose keys are on *both* the ActionErrors (and ActionMessages) *and* the JSP page will be displayed. Every other error message or error tag is ignored.

So, in Figure 8-3, only the error message for the property userid gets displayed. The error message for the property desc is ignored because there's no corresponding <html:errors> tag. The <html:errors> tag for the property named pwd doesn't display at all since there are no corresponding error messages for that property on the ActionErrors object.

■ **Tip** A useful trick is to leave out the property attribute like so: <html:errors/>. This causes *all* error messages to be displayed. This is especially useful for debugging.

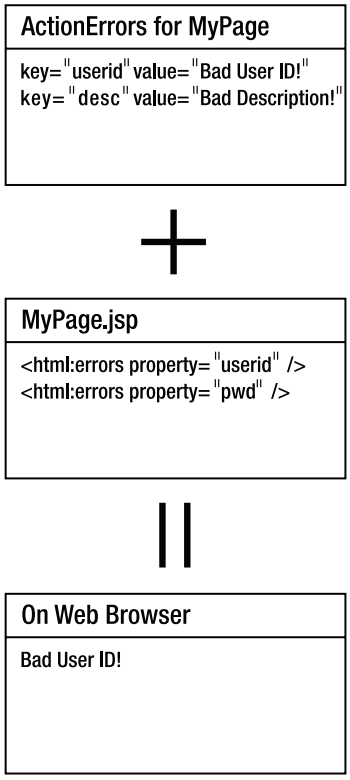

Figure 8-3. *Displaying errors*

To display a generic error message, not connected to any real form property, you can use a special *global property*. In your `ActionForm` or `Action`, you'd use the key `ActionMessages.GLOBAL_MESSAGE`:

```
errors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(...));
```

and the property value `org.apache.struts.action.GLOBAL_MESSAGE` on the `<html:errors>` tag:

```
<html:errors property="org.apache.struts.action.GLOBAL_MESSAGE"/>
```

Synopsis of HTML and Bean Tag Libraries

The previous section describes all the tags you'll use 70 percent of the time in your Struts webapps. The remaining 30 percent we'll describe in Chapters 10, 11, and 14. Appendix C is a comprehensive reference for *all* tags.

At this point, a synopsis of the HTML and Bean tags would be useful to you. Table 8-1 lists tags on the HTML library and their purpose. Table 8-2 does the same for the Bean tag library. You can find the details on each tag in Appendix C. Tables 8-1 and 8-2 are based on documentation made available by the Apache Software Foundation. The Apache License is available at `http://www.apache.org/licenses/LICENSE-2.0`.

Table 8-1. *Synopsis of the HTML Tag Library*

Tag	Usage/Comments
`base`	Generates an HTML `<base>` tag. This creates a reference from which all relative paths in your JSP will be calculated.
`html`	Generates an `<html>` tag. Also includes language attributes from the user's session.
`xhtml`	Tells other tags on the page to render themselves as XHTML 1.0–conformant tags.
frame	Generates an HTML `<frame>`.
`javascript`	Indicates the placement of autogenerated JavaScript. Used in conjunction with the Validator framework, described in Chapter 15.
`form`	Defines a form. The `action` and `focus` attributes are the most useful, followed by the `enctype` property described in Chapter 11.
`checkbox`	Generates a check box input field.
`file`	Generates a file select input field.
`hidden`	Generates a hidden field.
`multibox`	Generates a number of check box input fields. Used in conjunction with indexed properties.
`option`	Generates a select option.
`options`, `optionsCollection`	Generates a list of select options.
`password`	Generates a password input field.
`radio`	Generates a radio button input field.
`select`	Generates a select element.
`text`	Generates a textual input field.
`textarea`	Generates an HTML `textarea` element.
`image`	Generates an image input field.
`button`	Generates a button input field.
`cancel`	Generates a cancel button.
`submit`	Generates a submit button.
`reset`	Generates a reset button.
`errors`	Displays error messages.
`messages`	An iterator for error messages and ordinary messages.
`img`	Generates an HTML `img` tag.
`link`	Generates a hyperlink.
`rewrite`	Expands a given URI. Useful in creating URLs for input into your JavaScript functions.

Table 8-2. *Synopsis of the Bean Tag Library*

Tag	Usage/Comments
`message`	Writes static text based on the given key.
`write`	Writes the value of the specified JavaBean property.
`cookie/header/parameter`	Each defines a scripting variable based on the specified cookie/header/parameter.
`define`	Defines a scripting variable based on the specified JavaBean.
`page`	Exposes a page-scoped variable as a JavaBean.
`include`	Allows you to call an external JSP or global forward or URL and make the resulting response data available as a variable. The response of the called page is not written to the response stream.
`resource`	Allows you to read any file from the current webapp and expose it either as a String variable or an InputStream.
`size`	Defines a new JavaBean containing the number of elements in a specified `Collection` or `Map`.
`struts`	Exposes the specified Struts internal configuration object as a JavaBean.

Lab 8: Contact Entry Page for LILLDEP

In this lab, you will complete the implementation for the main page for LILLDEP. This page is called `full.jsp`.

1. Open `\lilldep\web\full.jsp` in the editor. Put in the missing taglib declarations. Which libraries would you use? (The TLD files are in `\web\WEB-INF\`.)

2. Modify `web.xml` to include the tag library locations.

3. Implement the tags for the missing properties. The properties for this form are listed in Lab 6. Use the keys found in `\web\resources\Application.properties` to display any messages you might need.

4. Put in error tags corresponding to each validated field in Lab 6. Also put in an error tag for the error encountered when saving a contact in Lab 7.

5. Implement the submit button. What is the name of the handler for this form?

■ **Note** The source code answers are in the `answers` folder in the LILLDEP distribution. The answers to questions raised in this lab are found in Appendix D.

Useful Links

- FreeMarker, a powerful open source template engine: `http://freemarker.sourceforge.net`
- Velocity, a simpler, lightweight open source template engine from Apache: `http://jakarta.apache.org/velocity/`
- *Pro Jakarta Velocity: From Professional to Expert*, by Rob Harrop (Apress, 2004): `www.apress.com/book/bookDisplay.html?bID=347`

Summary

In this chapter, you have seen how the basic Struts tags work to display form data and error messages.

- The Struts tags to display form data and error messages are in the HTML and Bean libraries.
- Forms have an associated form handler, which is a particular combination of `ActionForm` and `Action` subclasses.
- Form data is first passed through simple validation (`ActionForm`) before business logic is processed (`Action`).
- Struts handles the redisplay of badly filled forms and error messages for the necessary fields.

CHAPTER 9

Configuring Struts

Up until now, I've presented portions of Struts along the lines of the MVC design pattern. In Labs 6, 7, and 8, you've implemented these portions yourself for the LILLDEP webapp. What's missing is the way Struts ties together the various Model-View-Controller portions.

Struts uses a single configuration file called `struts-config.xml` to store this information. The Struts distribution comes bundled with samples of this file, which you can copy and amend for your own use.

The Structure of struts-config.xml

Unsurprisingly, `struts-config.xml` holds data in XML format. There are several sections, each of which handles configuration for specific portions of Struts:

- **Form bean declarations**: This is where you map your `ActionForm` subclass to a name. You use this name as an alias for your `ActionForm` throughout the rest of the `struts-config.xml` file, and even on your JSP pages.
- **Global exceptions**: This section defines handlers for exceptions thrown during processing.
- **Global forwards**: This section maps a page on your webapp to a name. You can use this name to refer to the actual page. This avoids hardcoding URLs on your web pages.
- **Form handlers**: Remember the form handlers I mentioned in the previous chapter? This is where you declare them. Form handlers are also known as "action mappings."
- **Controller declarations**: This section configures Struts internals. Rarely used in practical situations.
- **Message resources**: This section tells Struts where to find your properties files, which contain prompts and error messages.

- **Plug-in declarations**: This is where you declare extensions to Struts. You will use two important plug-ins: **Tiles** and **Validator**.

These sections must be placed in this order. Not all sections need be present. For example, you might not want to take advantage of the global exception handling facility. In this case, your `struts-config.xml` file would not contain the global exceptions section.

■ **Note** Earlier versions of Struts had a "data sources" section before the form bean declarations. This data sources section allowed you to preconfigure JDBC data sources for your webapp. This section has since been deprecated.

Among the seven sections, the form bean, form handler (or action mapping), and message resources sections are the most important, and you must master them before you can use Struts.

Before I introduce the various sections, let's take a look a simple `struts-config.xml` file, the one for the Registration webapp.

Configuring the Registration Webapp

The `struts-config.xml` file (shown in Listing 9-1) for the Registration webapp requires just the form bean, global exceptions, global forwards, form handler (or action mapping), and message resources sections.

Listing 9-1. *struts-config.xml for the Registration Webapp*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
   "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>

   <form-beans>
     <form-bean
        name="RegistrationForm"
        type="net.thinksquared.registration.struts.RegistrationForm"/>
   </form-beans>
```

```
  <global-exceptions>
    <exception key="reg.error.io-unknown"
               type="java.io.IOException"
               handler="net.thinksquared.registration.ErrorHandler"/>

    <exception key="reg.error.unknown"
               type="java.lang.Exception"
               path="/errors.jsp" />
  </global-exceptions>

  <global-forwards>
    <forward name="ioError" path="/errors.jsp"/>
  </global-forwards>

  <action-mappings>
    <action
       path="/Registration"
       type="net.thinksquared.registration.struts.RegistrationAction"
       name="RegistrationForm"
       scope="request"
       validate="true"
       input="/Registration.jsp">

        <forward name="success" path="/Success.jsp"/>

    </action>
  </action-mappings>

  <message-resources parameter="Application"/>

</struts-config>
```

The three main sections should be immediately obvious in Listing 9-1: the form beans section is enclosed by the `<form-beans>` tag, the form handlers section by the `<action-mappings>` tag, and the single message resource section by the `<message-resources>` tag. I'll describe these sections in detail next.

Declaring Form Beans

The form bean section is where you give your `ActionForm` subclasses names, which can be used both in `struts-config.xml` and on your JSP pages.

The declaration consists of a single enclosing <form-beans> tag (note the plural), and one or more <form-bean> (note the singular) tags, as shown in Listing 9-2.

Listing 9-2. *The Form Beans Section*

```
<form-beans>
  <form-bean
       name="RegistrationForm"
       type="net.thinksquared.registration.struts.RegistrationForm"/>
</form-beans>
```

A <form-bean> tag has two attributes:

- name: The name attribute is a unique label for an ActionForm subclass. In Listing 9-2, the name is RegistrationForm, but it could be anything you like, as long as it is unique among other form bean declarations and qualifies as an XML attribute value.
- type: The type attribute is the fully qualified class name for that ActionForm subclass.

With this declaration, you can use the name RegistrationForm to refer to the ActionForm subclass, net.thinksquared.registration.struts.RegistrationForm.

Declaring Global Exceptions

Global exceptions allow you to catch uncaught runtime exceptions that occur in your Action subclasses, displaying them with a custom error message. This certainly lends a little more polish to your application, compared to the default Tomcat error page. Listing 9-3 shows two ways you can define an exception handler.

Listing 9-3. *Declaring Global Exception Handlers*

```
<global-exceptions>
  <exception key="reg.error.io-unknown"
             type="java.io.IOException"
             handler="net.thinksquared.registration.IOErrorHandler"/>

  <exception key="reg.error.unknown"
             type="java.lang.Exception"
             path="/errors.jsp"/>

</global-exceptions>
```

The <global-exceptions> tag contains global exception handlers, each represented by an <exception> tag. Each <exception> tag has two required attributes:

- key: An error key. When an exception handler is fired, an ActionMessage with this key is created and put on the request. This error message gets pasted on the JSP containing an <html:errors> tag that is finally displayed. Note that key is a required attribute, which you have to specify even if you don't use it.

- type: Describes the type of error that is caught. Struts will first try to find the error class that matches the declared types. If none matches exactly, then Struts goes up that error's superclass tree, until it finds a match with a declared exception.

There are two ways you can handle the caught exceptions: using a path attribute to point to a JSP page that displays the error message, or using a handler attribute, which is a fully qualified class name of your ExceptionHandler subclass—that is, your handler subclasses:

```
org.apache.struts.action.ExceptionHandler
```

You only have to override the execute() function:

```
public ActionForward execute(Exception ex,
                             ExceptionConfig ae,
                             ActionMapping mapping,
                             ActionForm form,
                             HTTPRequestServlet request,
                             HTTPResponseServlet response)
```

As you can see, this is nearly identical to Action.execute(), apart from the extra first two arguments. You should declare the "next" page ins a <global-forwards> section (see the next section), so that mapping can resolve it.

In most instances, using a custom ExceptionHandler subclass is overkill unless you have a special requirement to do so.

Declaring Global Forwards

You use global forwards to define forwarding paths accessible to all Actions or ExceptionHandlers. In fact, all such paths will be accessible as long as you have an ActionMapping instance initialized by Struts. This is certainly the case for the execute() functions in Action or ExceptionHandler.

Global forwards are defined within an enclosing <global-forwards> tag, within which you may place as many <forward> tags as you wish, as shown in Listing 9-4.

Listing 9-4. *Declaring a Global Forward*

```
<global-forwards>
   <forward name="ioError" path="/errors.jsp"/>
</global-forwards>
```

Each <forward> tag has two attributes:

- name: A globally unique name for this forward.
- path: The path to the JSP or form handler to which this forward refers. In both cases, the paths must begin with a slash (/).

In Listing 9-4, a forward named ioError is declared, which refers to the path errors.jsp. You can access this path from the mapping object available in execute():

```
mapping.findForward("ioError")
```

Declaring Form Handlers

Form handlers are defined within a single <action-mappings> enclosing tag, as shown in Listing 9-5.

Listing 9-5. *The Form Handler Declaration*

```
<action-mappings>
  <action
    path="/Registration"
    type="net.thinksquared.registration.struts.RegistrationAction"
    name="RegistrationForm"
    scope="request"
    validate="true"
    input="/Registration.jsp">

     <forward name="success" path="/Success.jsp"/>

  </action>
</action-mappings>
```

The <action-mappings> tag acts as a container for each form handler, described by an <action> tag. The <action> tag contains a few attributes that configure the handler:

- path: Describes the name of the form handler.
- type: The fully qualified class name of the Action subclass that handles business logic.
- name: The name of the form bean associated with this handler.
- validate: Tells Struts whether it should perform simple validation on the form data.
- scope: Sets the scope of the form data. Only request or session scopes are allowed.
- input: The relative path of the page that forms the input of this page.

The path attribute must start with a slash. This has a meaning I will cover in a later chapter. You might want to recall how form handlers are used on your JSPs, as the action attribute of the <html:form> tag:

```
<html:form action="Registration.do" ...>
```

In this snippet, the form handler is Registration, and this corresponds to the path attribute in the form handler declaration of Listing 9-5. The .do extension tells the servlet container that this is a Struts form handler. This extension is the default, and is defined in the web.xml file distributed with Struts.

Just as in the form bean declaration, the type attribute is the fully qualified name of your Action subclass that will process business logic. In this case, it is RegistrationAction.

The name attribute is the name of the form bean associated with the handler. In this case, it is RegistrationForm, which is an obvious alias to the RegistrationForm subclass.

The validate : attribute, which can either be true or false, tells Struts whether the validate() method on the form bean's ActionForm subclass should be called. In other words, Struts performs simple validation only when validate="true".

The scope : attribute, which can either be request or session, tells Struts in which scope to put the form bean that contains the form data.

Lastly, the input attribute is just the path to the input page. This is how mapping.getInputForward() (see Chapter 7) knows what the input page was. In Listing 9-5, the input page is Registration.jsp.

Note that the path to the input page must begin with a slash, which denotes the "root" directory of the webapp. This corresponds to the root directory on the WAR file you create to deploy the webapp. Or, as another way of looking at it, when the webapp is installed, the root directory is just the \webapps\<app name>\ directory hosted under the servlet container.

For example, if the Registration webapp were deployed in a WAR file called registration.war, then the root directory on the servlet container would be \webapps\registration\.

One last thing about the `input` attribute. If I had put in a global forwards section, I could have declared a global forward for `Registration.jsp`, and used the name of the global forward as the value of `input`.

Forwards

Each `<action>` tag can contain zero or more `<forward>` tags. These forwards differ from the `<forward>`s declared in the global forwards section in that they are visible only to the enclosing form handler. Forwards declared the global forwards section are visible everywhere.

`<forward>`s represent the possible "next" pages for that form handler. Like the `<forward>`s in the global forwards section, each `<forward>` tag has two attributes:

- `name`: A name for the "next" page
- `path`: The actual page's path, relative to the webapp's root directory

The `name` attribute identifies this "next" page in the `Action` subclass associated with the form handler. This is how `mapping.findForward()` works (see Listing 7-2):

```
return mapping.findForward("success");
```

The label `success` is just the value of the `name` attribute.

The `path` must start with a slash if it's a true path. If you had declared the page as a global forward, then you could use the name of the global forward instead. In this case, you'd have no initial slash, because the `path` value is just a label, not a true path.

Controller Declaration

The controller section is probably the least used among the seven sections. It is used to manually override some default Struts settings. I will only mention a few useful ones here:

- `maxFileSize`: Specifies the upper size limit on uploaded files. You can use a number followed by `K`, `M`, or `G` to indicate a size in kilobytes, megabytes, or gigabytes, respectively. For example, `maxFileSize="2M"` limits the file size to 2MB. We will deal with file uploading in Chapter 11.
- `nocache`: Tells Struts whether it should cache content. Setting `nocache="true"` disables content caching.
- `contentType`: Specifies the default content type of pages. For example, if you're delivering XML pages by default, set `contentType="text/xml"`.

- The following snippet shows how you'd declare a controller section:

```
<controller maxFileSize="1.618M" contentType="text/svg" />
```

This sets the maximum uploadable file size to a golden 1.618MB, and the default content type to SVG.

Message Resources

The message resources section declares the location of your properties file that stores the key/value pairs of your application. Recall that the contents of this properties file was implicitly used to create error messages:

```
errors.add("userid",new ActionMessage("reg.error.userid.exists"));
```

and prompts on the JSP page:

```
<bean:message key="registration.jsp.prompt.userid"/>
```

Unlike the previous two sections, this one does *not* have an enclosing tag.

```
<message-resources parameter="Application"/>
```

The main attribute of the `<message-resources>` tag is the `parameter` attribute, which gives the location of the properties file for your application relative to the `\WEB-INF\classes\` directory of your webapp. So, in the previous declaration, the message resource file is

```
\WEB-INF\classes\Application.properties
```

Note that in the declaration, the `.properties` extension is implied. If you had placed the properties file further up the package, say, in

```
\WEB-INF\classes\net\thinksquared\registration\struts\resources\
```

the `parameter` attribute value would be

```
net.thinksquared.registration.struts.resources.Application
```

Declaring Plug-ins

Plug-ins are custom extensions to Struts. An example is the Tiles framework, which was developed independently of Struts. In earlier (pre-1.2) versions of Struts, Tiles had to be downloaded separately. Although it is now part of the 1.2 distribution, its earlier independent existence is still apparent because you have to declare a plug-in section for Tiles in order for you to use it in Struts.

The plug-in section tells Struts what plug-ins to initialize and what data they require (usually paths to various configuration files required by the plug-in). Listing 9-6 shows a typical plug-in declaration.

Listing 9-6. *A Possible Plug-in Declaration for Tiles*

```
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
    <set-property property="definitions-config"
                    value="/WEB-INF/tiles-defs.xml"/>
</plug-in>
```

Each `<plug-in>` tag declares a single plug-in. There is no limit to how many of these you might have in your `struts-config.xml` file. The `className` attribute is required, and it points to the plug-in class that Struts calls. This class is unique to each plug-in.

Each `<plug-in>` tag may contain zero or more `<set-property>` tags to set the various properties needed by the plug-in. There are only two attributes here:

- `property`: Defines the property being set
- `value`: Specifies the corresponding value

Needless to say, each plug-in will have to be configured differently, and you'll have to get the details from the plug-in's manual.

Lab 9a: Configuring LILLDEP

In this lab, you will configure LILLDEP and deploy it on Tomcat. Make the following changes to `\web\WEB-INF\struts-config.xml` for LILLDEP:

1. Declare a form bean called `ContactFormBean` for the `ActionForm` subclass you implemented in Lab 6.
2. Declare a form handler for the form in `full.jsp`. What should the name of the form handler be?
3. The `Action` subclass for the form handler should be the one you implemented in Lab 7. The form bean used should be the one in step 1.
4. Set the `scope` attribute to `request`.
5. Give a value for the form handler's `input` attribute. Where is this used? What happens if you omit this attribute from the form handler's declaration?

6. Create a forward for this form handler to the page `full.jsp`. What should be the `name` attribute of this forward? (Hint: Check the code for the `Action` subclass.) What happens if you omit this forward declaration?

7. Run `compile.bat` to produce the WAR file, then deploy and test your application. Figure 9-1 shows what the LILLDEP start page should look like. You should be able to key data into LILLDEP. Test out the simple validations you wrote in Lab 6.

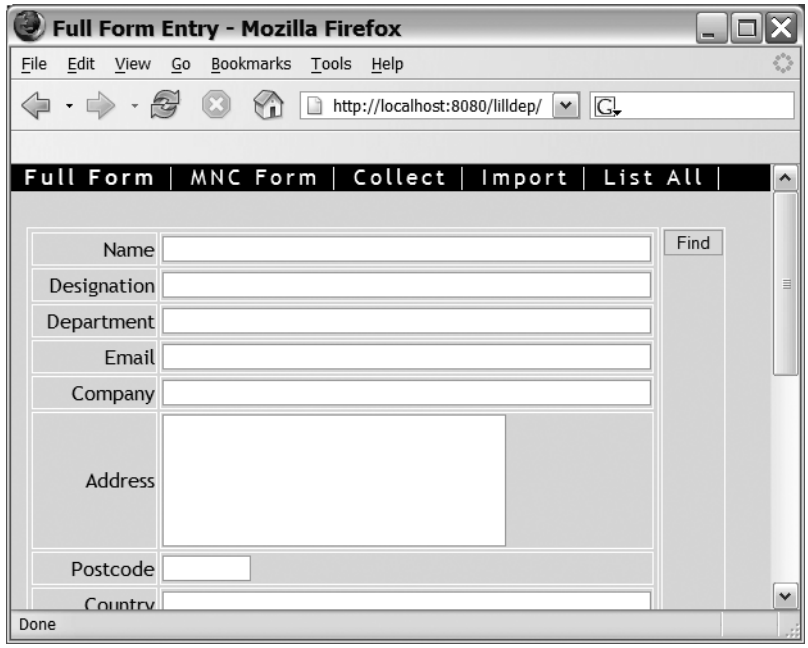

Figure 9-1. *The LILLDEP start page*

Code Reuse

One reason why form handlers are declared in the `struts-config.xml` file instead of being hardcoded is because this promotes code reuse.

In Lab 9a, you implemented a form handler specific to the `full.jsp` page. But what if you wanted a different page to submit data to this form handler? You obviously can't now, because this form handler's only "next" page is `full.jsp`.

One way to do it would be to declare a new `<forward>`, and amend `ContactAction` to dispatch to the forward corresponding to the input page. This "solution" is hardly a good one because you'd have to amend and recompile `ContactAction` each time you wanted to add or remove a page.

The solution Struts offers is much better. The idea is to declare a *new* form handler in `struts-config.xml` to handle input from a different page. You can then reuse form beans and `Action` subclasses in this new form handler, but use a different `<forward>`.

The next lab session shows you how to do this.

Lab 9b: The MNC Page

In LILLDEP, users frequently have to fill in contacts from multinational corporations (MNCs). For these companies, users want a separate form, containing just the following seven fields:

- Name
- Designation
- Department
- Email
- Company
- Address
- Postcode

The Classification field should *automatically* be set to the string value `mnc`. You can use the `<html:hidden>` tag

```
<html:hidden property="myFormProperty" value="myFixedValue"/>
```

to automatically set the Classification field. The other fields should be left blank. Figure 9-2 shows how the MNC page appears to the user.

1. Complete the implementation of `mnc.jsp`. The form handler should be called `ContactFormHandlerMNC`.
2. Add a new form handler to `struts-config.xml` to accept data from `mnc.jsp`. The (single) forward should point back to `mnc.jsp`. What should the name of the forward be? (Hint: `ContactAction` knows only one forward label.)
3. Run `compile.bat` to produce the WAR file.
4. Stop Tomcat, then delete the `\webapps\lilldep\` folder. Only then re-deploy the LILLDEP webapp.

Test out your application as you did in Lab 9a.

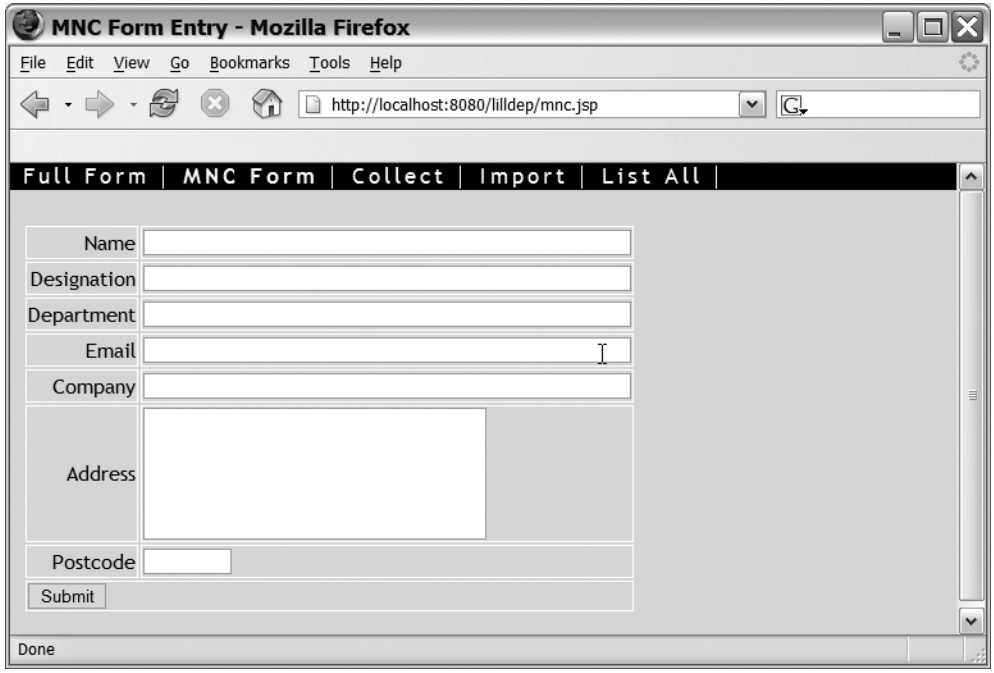

Figure 9-2. *The MNC data entry page*

Summary

In this chapter, you've learned the basics of configuring Struts:

- The `struts-config.xml` file ties together the View and Controller sections of your webapp.
- `struts-config.xml` is deployed in the `\WEB-INF\` directory of your webapp.
- `ActionForm` subclasses are exposed to your webapp in the form bean declarations.
- A form handler declaration ties together form beans, an `Action` subclass, and one or more "next" pages.
- The properties files for your application are declared in the message resources section of `struts-config.xml`.

CHAPTER 10

More Tags

In Chapter 8 we covered the basic Struts tags: those used to display static text, to facilitate data entry on forms, and to display error messages.

The story as we take it up in this chapter becomes a little more complex. Since Struts was first developed, a couple of paradigm shifts have occurred in the Java/JSP world: the JSP Standard Template Library (JSTL) and JavaServer Faces (JSF).

JSTL is an attempt to create a set of standardized JSP tags to perform certain generic tasks, such as iteration, conditional processing, formatting, internationalization, XML processing, and data access. Prior to the advent of JSTL, each JSP-related technology like Struts had to create custom tags, from scratch, to do these things.

This do-it-yourself approach has two big drawbacks:

- **A steeper learning curve**: Developers new to such a JSP-related technology would have to learn how to perform generic things using that technology's custom tags.

- **Incomplete or inflexible tag libraries**: Because the thrust of the new technologies was *not* to do generic things well, it meant that these custom tag libraries were often lacking in flexibility and power. Developers would often have to resort to ugly tricks with scriptlets to get the functionality they needed.

JSTL solves these problems by providing tags to perform many of these generic functions well. It also makes heavy use of a "mini-language" called the expression language (EL) (from the new JSP 2.0 specification), which makes these tags very flexible. For these reasons, JSTL is the preferred technology to use when it overlaps with custom Struts tags.

Unfortunately, JSTL requires servlet containers that conform to the JSP 2.0 specification in order to work. This is a problem for those who have to work with "legacy" servlet installations that are not JSP 2.0 conformant.

The other parallel development is JavaServer Faces (JSF), a specification by Sun, which (among other things) provides a set of custom tags to display a variety of client-side interfaces that far exceeds the paltry collection in the Struts HTML and Bean tag libraries. The Apache Shale project (more on this in Chapter 20) is a ground-up rework of Struts around the JSF specification. It is certainly possible that some of the tags from Shale will make their way into Struts in the future.

This chapter covers two more Struts tag libraries, and some of the relevant overlaps between Struts and JSTL. We'll also look at an effort to get Struts tags to accept EL expressions. We'll describe JSF and Shale in Chapter 20.

But first, here are a couple of guiding lights to lead the way.

Best Practices

It's easy to get lost in the morass of tags and tag libraries, so I'd like you to consider two rules of thumb I believe are useful in deciding which tag library to go with when you have a choice:

- **Beware the scriptlet**: Scriptlets are not inherently "evil," but if you find yourself using them to overcome deficiencies in a tag, it's time to consider scouting for an alternative.

- **Use "custom-built generic solutions"** (pardon the oxymoron): Use generic solutions like JSTL whenever possible.

I hope the rationale behind these two heuristics is obvious. Custom tags are there to obviate the need to use scriptlets. So, if you find yourself having to use scriptlets, the custom tags you're using aren't doing their job well enough.

The 2 + 1 Remaining Struts Libraries

In Chapter 8, you saw how to use the HTML and Bean tag libraries. There are three other Struts tag libraries: the **Logic**, **Nested**, and **Tiles** tag libraries.

■ **Note** In older versions of Struts, prior to 1.2, you might come across a "template" tag library. The Tiles library supersedes this older "template" tag library.

In this chapter, we'll only cover the Logic and Nested tag libraries. The Tiles tags warrant an entire chapter of their own (Chapter 14).

The Logic Tag Library

The Logic library has tags for

- **Iteration**: The `iterate` tag
- **Conditional processing**: The `equal`, `present`, `match`, `empty`, `notEqual`, `notPresent`, `notMatch`, `notEmpty`, `messagesNotPresent`, `messagesPresent`, `lessThan`, `lessEqual`, `greaterThan`, and `greaterEqual` tags
- **Flow control**: The `redirect` and `forward` tags

■ **Note** All of these tags, apart from the ones for flow control, have counterparts in JSTL, which we will cover later in this chapter. You should use the JSTL versions whenever possible.

Let's tackle these three classes of tags next.

Iteration

The `<logic:iterate>` tag can be used to iterate over collections, enumerations, arrays, or iterators. We'll refer to any of these as an "iteratable" object. To use the `<logic:iterate>` tag, you have to

1. **Create the iteratable object**: This is done in your `Action` subclass, since this is Controller code. Never forget the MVC.
2. **Place the iteratable object on the request**: Recall that in your `Action`'s `execute()` function, you have a reference to the request object. You can place objects on this request object, or on its associated session object, giving the object a *name*.
3. **Forward to the JSP page that contains the `<logic:iterate>` tag**: The JSP page called by the `Action` would be able to access these objects by the name you gave them.

These three steps are the same or similar even if you're using JSTL for iteration, so be sure to study carefully the following code listings that illustrate these steps.

We'll take up the LILLDEP webapp you've been working on throughout this book. The requirement is to create a simple search page for `Contacts` by postcode. The user keys in the postcode and the page returns a list of `Contacts`. This is obviously a contrived example, but it keeps things simple!

Also, for the sake of clarity, we'll present only the relevant, nonobvious portions of code. First, Listing 10-1 shows the `Action` subclass, aptly named `PostcodeSearchAction`, which receives the postcode the user wants to search by.

Listing 10-1. *A Section of `PostcodeSearchAction`*

```
public ActionForward execute(...){

   PostcodeSearchForm psForm = (PostcodeSearchForm) form;
   String postcode = form.getPostcode();

   // Step 1: create iteratable object
   // Retrieve a list of Contacts matching the given postcode.
   // If you find the following code confusing, read Appendix A.

   Criteria crit = new Criteria();
   crit.add(Contact.POSTCODE,postcode);
   Iterator results = ContactPeer.doSelect(crit);

   // Step 2: put iteratable on request
   // Save the list on the results. You might want to refer to
   // Appendix B for some the functions on HttpServletRequest.

   request.setAttribute("search_results", results);

   // Step 3: forward to the JSP
   // In this case, struts-config.xml maps "success" to
   // Display.jsp

   return new ActionForward(mapping.findForward("success"));
}
```

So, an `Iterator` of `Contacts` is located on the request object. This `Iterator` is called `search_results`. Listing 10-2 shows the section of `Display.jsp` that lists the `Contacts`.

Listing 10-2. *A Section of `Display.jsp`*

```
<logic:iterate name="search_results" id="contact" indexId="cnt">
     <p>
        Contact number <bean:write name="cnt"/>
        <bean:write name="contact" property="name"/>
        <bean:write name="contact" property="company"/>
        <bean:write name="contact" property="tel"/>
     </p>
</logic:iterate>
```

In Listing 10-2, the name, company, and telephone number of each `Contact` is listed. The `<logic:iterate>` tag has three main attributes:

- `name`: Refers to the iteratable object you've placed on the request. In this instance, it's `search_results`.
- `id`: Specifies a name that represents a single element on the iteratable object. In this case, the `id` is `contact`. This attribute is required, even if you don't use it.
- `indexId` (optional): Specifies a number representing the current iteration. The only reason I used it here is to write out the iteration number using `<bean:write>`. The `indexId` is important when you want to use *indexed properties*, a topic I will come to in the next section.

The `<bean:write>`s write out the name, company, and telephone of a given `Contact`. Note that the `name` attribute of `<bean:write>` refers to the single element exposed by the `id` attribute of `<logic:iterate>`. Refer to Appendix C for a thorough explanation of `<bean:write>`.

ITERATING OVER HASHMAPS

If the iteratable object is a `HashMap` or a class implementing a `Map`, the elements are instances of `Map.Entry`. This interface contains `getKey()` and `getValue()` functions, so you can use

```
<bean:write name="xxx" property="key"/>
```

to display the key, and

```
<bean:write name="xxx" property="value"/>
```

to display the corresponding value.

I'd like to reiterate (pardon the pun) that the functionality provided by `<logic:iterate>` overlaps with JSTL's `<forEach>` tag. We'll come to this in a later section.

Simple, Nested, Indexed, and Mapped Properties

Many Struts tags accept a `property` attribute, which refers to a property on the object they're operating on. We've seen this in action, for example, in the `<bean:write>` tag in the previous section:

```
<bean:write name="contact" property="company"/>
```

As it's used here, `<bean:write>` attempts to display the return value of `getCompany()`, which is called on the object referred to as `contact`. This is an example of a *simple property*.

Simple properties are those that implicitly call a `getXXX()` function on the object currently under consideration.

Struts supports three other types of properties: *nested*, *indexed*, and *mapped* properties. **Nested properties** allow you to refer to a property of a property. For example, if in our previous example the return value of `getCompany()` were a JavaBean object, then nested properties would allow you to refer to a property on this object:

```
<bean:write name="contact" property="company.name"/>
```

As it's used here, `<bean:write>` attempts to display the return value of `getCompany().getName()`.

Indexed properties are similar to simple properties, but they assume that the function that should be called is `getXXX(int index)` instead of just `getXXX()`. So, if the `contact` object supported a `getCompany(index)` function, this is how you would call it:

```
<bean:write name="contact" property="company[41]"/>
```

As it's used here, `<bean:write>` attempts to display the return value of `getCompany(41)`. This example begs the question of how to use indexed properties with `<logic:iterator>`, for example, to display a list of values. This is how you'd do it:

```
<logic:iterate name="companies" id="company" indexId="cnt">
   <bean:write name="companies" property="company[<%=cnt%>]"/>
</logic:iterate>
```

This example assumes that the `companies` object has a `getCompany(index)` function.

This example should also set off alarm bells in your head: it uses a scriptlet! There's got to be a better way—and there is. In this particular example, you have two choices: use `<forEach>` and `<out>` JSTL tags (which we'll cover later) or use the Struts-EL tags (we'll also cover these later), which are souped-up versions of the ordinary Struts tags:

```
<logic-el:iterate name="companies" id="company" indexId="cnt">
   <bean-el:write name="companies" property="company[${cnt}]"/>
</logic-el:iterate>
```

In this example, the change is hardly perceptible, but in more realistic cases the resulting code would be much simpler to read.

Mapped properties are similar to indexed properties, except that the function called is assumed to be `getXXX(String s)`. For example, if the `contact` object in the previous examples had a `getMobile(String mobileType)` function, you could use

```
<bean:write name="contact" property="mobile(home)"/>
<bean:write name="contact" property="mobile(office)"/>
```

These examples would call `getMobile("home")` and `getMobile("office")`, respectively.

You can also mix the various types of properties:

```
<bean:write name="contacts" property="contact[5772].company"/>
```

which is an implicit call to `getContact(5772).getCompany()`.

Table 10-1 summarizes the various property types.

Table 10-1. *Simple, Nested, Indexed, and Mapped Properties*

Type	Usage	Function Called
simple	property="myProperty"	getMyProperty()
nested	property="myProperty.mySubProperty"	getMyProperty().getMySubProperty()
indexed	property="myProperty[5772] "	getMyProperty(5772)
mapped	property="myProperty(myString) "	getMyProperty("myString")

Conditional Processing

The Logic tag library has a number of tags for performing conditional processing. These may be grouped into three categories:

- **Tests:** `present`, `empty`, `messagesPresent`
- **Comparisons**: `equal`, `lessThan`, `lessEqual`, `greaterThan`, and `greaterEqual`
- **String matching**: `match`

Some of these tags (for tests and string matching) have negative counterparts: `notPresent`, `notEmpty`, `messagesNotPresent`, and `notMatch`. All these tags follow a common structure in their usage. In pseudocode:

```
<tag>
  // conditionally processed code here.
</tag>
```

In addition, all these tags have two attributes in common:

- `name`: Specifies the name of the object that the tag operates on. It is assumed that a previous `Action` had placed the object on the request. The earlier section on iteration shows how this might be done.
- `property`: Corresponds to a simple, nested, indexed, or mapped property on the object referred to by the `name` attribute. This attribute is optional.

Tags that require a comparison or a match will also accept an additional `value` attribute. Let's look at a couple of examples of how these tags are used:

```
<logic:empty name="search_results">
   <i>Sorry, there are no contacts with this postcode</i>
</logic:empty>
<logic:notEmpty name="search_results">
   <!-- code to display contacts goes here -->
</logic:notEmpty>
```

The `empty` and `notEmpty` tags simply check if the given iteratable object contains elements, and therefore don't require a `value` attribute. Here's another example:

```
<logic:lessThan name="myConstants" property="pi" value="3.14">
   <i>Some of your pi is missing!</i>
</logic:lessThan>
```

This code displays the message "Some of your pi is missing" if `myConstants.getPi()` is less than 3.14.

Table 10-2 summarizes these tags.

Table 10-2. *Summary of the Conditional Processing Tags*

Tag	Meaning
`present`	Tests if a variable is present on the request.
`empty`	Tests if an array or `Collection` is empty.
`messagesPresent`	Tests if a message (an `ActionMessage` instance) is on the request. The name of the message is given in the `value` attribute.
Comparison tags (`equal`, `lessEqual`, etc.)	Tests if the given property passes the given comparison. The value to be compared with is provided in the `value` attribute.
`match`	Checks if the given property has the `value` attribute as a substring.

The negative counterparts of the tags in Table 10-2 are `notPresent`, `notEmpty`, `messagesNotPresent`, and `notMatch`). Appendix C describes all these tags in more detail.

Lastly, note that JSTL has an `<if>` tag that you can use in place of many of these tags. We'll come to this later on in this chapter.

Flow Control

The two tags that perform flow control (actually, redirection) are `<logic:redirect>` and the simplified version of it, called `<logic:forward>`. Their use on a JSP page causes control to be passed to a different page. For example:

```
<logic:forward name="someOtherPage"/>
```

would redirect control to a global forward (refer to Chapter 9) called `someOtherPage`.

`<logic:redirect>` also redirects control to a different page, but gives you more options for performing the redirect. For more details on each of these tags, refer to Appendix C.

Note that there are no equivalent JSTL tags that can utilize global forwards declared in `struts-config.xml`.

The Nested Tag Library

The Nested tag library allows you to apply tags *relative* to an object. In some instances, this can greatly simplify your server-side `ActionForm` code, as you'll see in the lab sessions of this chapter.

■ **Note** If you've programmed in Visual Basic, you might have used the `With` keyword. The Nested tag library is an inelegant implementation of this.

To illustrate how the Nested tag library works, let's first consider the class diagram shown in Figure 10-1.

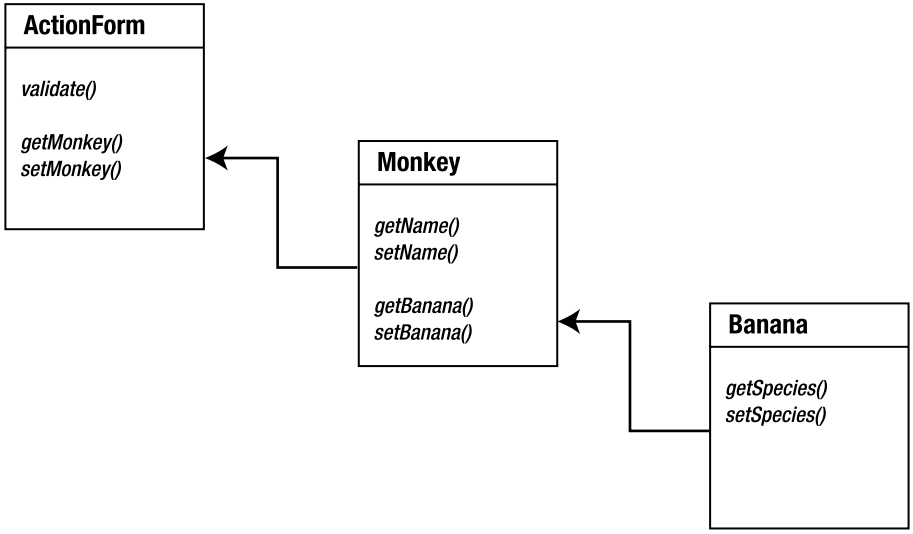

Figure 10-1. *The `MyActionForm`, `Monkey`, and `Banana` classes*

Figure 10-1 describes an `ActionForm` subclass and its functions. The data on `MyActionForm` is held by two objects: `Monkey`, which in turn has a `Banana` object. Our webapp has to allow the user to key in a name for the `Monkey`, and a species for the `Monkey`'s `Banana` (Figure 10-2).

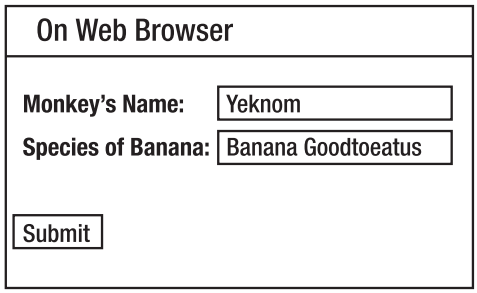

Figure 10-2. *Form to enter the Monkey's name and the Banana species*

As Listing 10-3 shows, the Nested tag library allows you to do this *without* having to include additional getters and setters for the `name` and `species` in `MyActionForm`.

Listing 10-3. *MonkeyPreferences.jsp*

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-nested" prefix="nested" %>
<html:html>
<body>
  <html:form action="MonkeyPreferencesForm.do">
  <nested:nest property="monkey">
    Monkey's name:
    <nested:text property="name" size="60" />
    <br>
    <nested:nest property="banana">
      Species of Banana:
      <nested:text property="species" size="60" />
    </nested:nest>
  </nested:nest>
    <html:submit/>
  </html:form>
</body>
</html:html>
```

The <html:form> tag implicitly sets the *root object* as MyActionForm. This means that all properties are calculated relative to the MyActionForm instance. The <nested:nest> tag

```
<nested:nest property="monkey">
    Monkey's name:
    <nested:text property="name" size="60" />
```

makes the property references relative to the monkey property instead. So the <nested:text>'s property tag is really monkey.name.

Your eyes should widen at the <nested:text>. Yes, that's right. The Nested tag library contains "nested" counterparts for *all* of the HTML tag library, most of the bean tag library, and all of the Logic tag library apart from the flow control tags. Appendix C lists the tags in the HTML, bean, and logic tag libraries supported by the Nested tag library.

Obviously, there's nothing you can't achieve with the Nested tag library that you can't already do with nested *properties* (see the earlier section). For example, Listing 10-4 shows how Listing 10-3 could be rewritten without using nested tags. In complicated scenarios, though, the use of the Nested tag library makes your code a little neater.

Listing 10-4. *MonkeyPreferences.jsp, Take 2*

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html>
<body>
  <html:form action="MonkeyPreferencesForm.do">
    Monkey's name:
    <html:text property="monkey.name" size="60" />
    <br>
    Species of Banana:
    <html:text property="monkey.banana.species" size="60" />
    <html:submit/>
  </html:form>
</body>
</html:html>
```

Lastly, there are times when you want to specify a different root object than the default one. You use the <nested:root> tag to do this:

```
<nested:root name="myNewRoot">
   <!-- new root object is myNewRoot -->
</nested:root>
```

Of course, this example assumes that the object named myNewRoot has been placed on the request. This is usually done in your Action subclass. Refer to Listing 10-1 and the accompanying discussion to see how this is done.

JSTL and Struts

As we mentioned earlier, JSTL was created to give JSP developers a comprehensive set of tags to perform a variety of generic tasks. The ultimate goal is to allow JSP developers to avoid using scriptlets altogether.

You might ask what makes JSTL a more compelling alternative to scriptlets. Consider Listing 10-5 and then look at Listing 10-6, the JSTL version.

Listing 10-5. *Counting to 99*

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<body>
   <%
      for(int i = 1; i <= 99; i++){
   %>
         This is iteration number <%= i %>
   <%
      }
   %>
</body>
</html>
```

Listing 10-6. *Counting to 99 with JSTL*

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
<body>
   <c:forEach var="i" begin="1" end="99" step="1">
         This is iteration number <c:out value="${i}"/>
   </c:forEach>
</body>
</html>
```

Even in this simple example it's obvious that the JSTL example (Listing 10-6) is much more readable than Listing 10-5, which uses scriptlets.

Now, like I said, JSTL is a huge body of useful tags and covering them effectively would require an entire book.

■ **Note** You could check out *Pro JSP 2, 4th Edition* (Apress, 2005). This book covers JSP 2.0, JSTL, and much more.

What I will discuss here is the *intersection* between JSTL and Struts tags. Fortunately, this intersection is manageable. JSTL consists of four tag libraries: the Core, Formatting, XML Processing, and Database Access tag libraries. Of these, only the Core and Formatting tag libraries fall into the intersection. From these, only a few tags concern us. They fall into three categories:

- **Output**: Performed by the `<c:out>` tag. In Struts, you'd use `<bean:write>`. There are also a number of tags from the Formatting tag library to format text and display resource messages.
- **Iteration**: Performed by the `<c:forEach>` tag. The analogous Struts tag is `<c:iterate>`.
- **Conditional processing**: Performed by the `<c:if>` and `<c:choose>...<c:when>` tags. These have a lot of tags from the Struts Logic library as counterparts.

Before I describe these three JSTL tags, let's take a look at the expression language (EL). Much of the power of JSTL is attributable to the use of EL within its tags.

■ **Note** Although I'm introducing EL here with JSTL tags, this is for purely pedagogical reasons. EL is independent of JSTL and is part of the JSP 2.0 specification.

Expression Language (EL)

EL allows the attributes of "EL-enabled" tags to assume dynamic values. Before EL, if you wanted dynamic attributes, you had to embed scriptlets or custom tags within the attribute (see Listing 10-7), resulting in difficult-to-read code.

Listing 10-7 illustrates a typical situation where dynamic attributes are important and how you might use scriptlets to achieve this functionality. The snippet alternately colors the rows of a table gray, using a (fictitious) custom tag, `<h:tr>`, to write out HTML table rows with some fancy formatting.

Listing 10-7. *Alternately Coloring the Rows of a Table Gray*

```
<% for(int i = 0; i < 10; i++){  %>
   <h:tr bgcolor="#<%= (i%2 == 0)? 'FFFFFF' : 'DDDDDD' %>" >
      <td> This is row number <%= i %> </td>
   </h:tr>
<% } %>
```

Unless you're a Perl programmer, I'm sure you'll agree that Listing 10-7 is difficult to read. EL solves this problem by allowing EL-enabled tags to have dynamic attributes.

For example, if <h:tr> were an EL-enabled tag, then Listing 10-7 could be rewritten as shown in Listing 10-8, which is more readable.

Listing 10-8. *Alternately Coloring the Rows of a Table Gray, with EL and JSTL*

```
<c:forEach var="i" begin="0" end="9" step="1">
   <h:tr bgcolor="#${(i%2 == 0)? 'FFFFFF' : 'DDDDDD'}">
      <td> This is row number <c:out value="${i}"/> </td>
   </h:tr>
</c:forEach>
```

Now, EL isn't just scriptlets with `${...}` delimiters instead of <%...%> delimiters. EL includes several keywords and operators that aid in its role of endowing tags with dynamic attributes. We'll take a look at this next.

Using EL

EL expressions are evaluated within the `${...}` delimiters. Within an EL expression, you can use a number of common operators:

- **Arithmetic operators**: `+`, `-`, `%` or `mod`, `*`, `/` or `div`
- **Logical operators**: `and` or `&&`, `or` or `||`, `not` or `!`
- **Comparisons**: `<` or `lt`, `<=` or `le`, `==` or `eq`, `!=` or `ne`, `>` or `gt`, `>=` or `ge`
- **Conditional evaluation**: `(condition) ? A : B` returns `A` if `condition` is `true` and `B` otherwise. `A` or `B` may be fixed strings (e.g., `a string`) or a variable or numeric value. However, to make your code readable, keep it simple.
- `empty` **operator**: Prefix `empty` to the name of an object to determine if it is empty (an array, `Collection`, `Enumeration`, `HashMap`, or `Iterator`) or `null` (any other object).

These operators should require no further explanation, since they are similar to those in Java.

You may also use *nested* and *indexed* properties with EL. Nested properties are the same as for Struts: the `.` separator in a name indicates a nested property. For example:

```
${myObject.myProperty}
```

returns the value of `myObject.getMyProperty()`. No surprises here.

Indexed properties with EL do double duty. You can index with a numeric value, a string, or a variable. Indexing with a string is the same as using a mapped property. Table 10-3 offers some examples.

Table 10-3. *Indexed Properties and EL*

EL Expression	Value
`${myArray[2718].name}`	`myArray[2718].getName()`
`${myMap['exp'].value}`	`myMap.get("exp").getValue()`
`${myArray[myOffset]}`	Returns the element on `myArray` with an index equal to the variable `myOffset`.

Lastly, EL exposes several *implicit objects*, which are built-in objects that your EL expressions will always have access to. There are a quite few of these, and I won't go through every one. To learn more, refer to Sun's J2EE tutorial (see "Useful Links").

■ **Note** A common pitfall is to use a variable name coinciding with the name of an implicit object. For example, calling a variable of your own "requestScope" is a no-no. There are also a number of reserved keywords that you must avoid. These are given in Sun's tutorial (see "Useful Links").

Among the most important implicit objects are those representing the various scopes. These allow you to explicitly access your variables by scope. These scope objects are `pageScope`, `requestScope`, `sessionScope`, and `applicationScope`. Suppose you want to access a session-scoped variable named `myObject`. Here's how:

```
${sessionScope.myObject}
```

Variables on other scopes are accessed in a similar manner.

In this section, we'll examine the use of EL in the JSTL tags `<forEach>`, `<if>`, `<choose>...<when>`, and `<out>`, which are relevant to Struts. I hope that after you read this section you will have sufficient exposure to EL to be able to use it well.

The <c:out> Tag

The `<c:out>` tag outputs text, and is in many instances a replacement for `<bean:write>`. The usage of this tag is simple, since it has just one `value` attribute:

```
<c:out value="${myObject.myProperty}">
```

A quick look at Appendix C will show you that `<bean:write>` isn't a one-trick pony—it has attributes you can use to format the displayed value. You can't do this with just `<c:out>`; you'd have to use the JSTL Formatting tag library. Using this tag library requires a more thorough treatment of JSTL than can be done here. Again, check out *Pro JSP 2, 4th Edition,* if you're interested in learning more.

The <c:forEach> Tag

The `<c:forEach>` tag is used for iteration. You can use it instead of the Struts `<logic:iterate>` tag. You've seen how to use `<c:forEach>` to perform iteration over a fixed set of limits (see Listings 10-6 and 10-8). You can also use it to perform iteration over a Java array, `Collection`, `Enumeration`, or `Iterator`. Listing 10-9 shows you how to do this.

Listing 10-9. *Using `<c:forEach>` to Iterate Over an Iteratable Object*

```
<c:forEach var="item" items="${sessionScope.myIteratable}">
   <c:out value="${(empty item)? 'nil' : item.myProperty}"/>
</c:forEach>
```

The two important attributes here are `var`, which exposes a single element of the iteratable object, and `items`, which is the name of the iteratable object. Note how Listing 10-9 uses the `myIteratable` object, specifically with session scope, using the `sessionScope` implicit object.

Flashback Quiz

Can you remember what the counterparts of `var` and `items` were on `<logic:iterate>`?

There is an additional useful attribute called `varStatus`, which gives you information on the current iteration. `varStatus` has four properties:

- `first` and `last`, which return `true` or `false` depending on whether the current iteration is the first or last one in the loop.
- `count`, which gives you the number of the current iteration. This starts at 1.
- `index`, which gives you the absolute index of the current iteration. This starts at 0, and is incremented according to the value of the `step` attribute.

Listing 10-10 shows `varStatus` in action.

Listing 10-10. *The `first`, `last`, `count`, and `index` Properties of `varStatus`*

```
<c:forEach begin="10" end="15" step="2" varStatus="status">
   <c:out value="${status.first}"/>
   <c:out value="${status.last}"/>
   <c:out value="${status.count}"/>
   <c:out value="${status.index}"/><br>
</c:forEach>
```

The output of Listing 10-10 would be

```
true  false 1 0
false false 2 2
false true  3 4
```

The last two JSTL tags we'll look at are <c:if> and <c:choose>...<c:when> for conditional processing.

The <c:if> and <c:choose>...<c:when> Tags

<c:if> allows you to process a single block of code if a test passes. The test itself is an EL expression referred to by the test attribute, as shown in Listing 10-11.

Listing 10-11. *<c:if> in Action*

```
<c:if test="${(empty user.userid || empty user.password)}">
   Please key in your User ID and password.
</c:if>
```

In Listing 10-11, the message is displayed if the user ID or password are null. The <if> tag doesn't support an else or if else block. If you want these, then you have to use the clumsier <choose> ... <when> tags (see Listing 10-12). If you've used Extensible Stylesheet Language (XSL), this should be very familiar.

Listing 10-12. *<c:choose>...<c:when> in Action*

```
<c:choose>
   <c:when test="${empty user.userid}">
      Please key in your userid.
   </c:when>
   <c:when test="${empty user.password}">
      Please key in your password.
   </c:when>
   <c:when test="${not user.loggedIn}">
      Sorry, couldn't log you in.
   </c:when>
   <c:otherwise>
      You're logged in!
   </c:otherwise>
</c:choose>
```

In Listing 10-12 there are two when clauses that test if the user ID and password are null, and another that tests whether the user is logged in. The last otherwise tag is a catchall block. Note that the otherwise block is optional.

Now You Do It

Earlier in this chapter, I mentioned that some of the conditional processing tags have JSTL equivalents. Construct JSTL equivalents for

- `equal`, `present`, `empty`, `lessThan`, `lessEqual`, `greaterThan`, and `greaterEqual`
- The negatives for all of the above (`notEqual`, `notPresent`, etc.)

Refer to Appendix C if you need definitions for these tags. Appendix D contains the answers.

This section ends my treatment of JSTL. In the next section, we'll describe an extension to the original tag libraries that allows them to process EL.

Struts EL Extensions

In order for EL to work in a custom tag, that tag has to be EL enabled. The current set of Struts tags aren't EL enabled, so this won't work:

```
//won't work!!
<bean:write name=${(empty obj)? obj2.prop : obj.prop />
```

A set of EL-enabled Struts tags is available with the latest Struts distribution. You'll find these new tags in the `./contrib/struts-el/lib/` folder of the distribution. Also available in this folder is an implementation of JSTL. If you use these Struts EL tag libraries, you *can* use EL expressions in your Struts tags.

Of course, you'd have to use the corresponding Struts-EL tag, not the original Struts tag. The following snippet illustrates this:

```
//OK
<bean-el:write name=${(empty obj)? obj2.prop : obj.prop />
```

Of course, just changing the prefix won't work—you'd have to change the taglib declaration in the JSP and put in a new `<taglib>` section in `web.xml` to point to the Struts-EL bean tag library.

The Struts-EL tag libraries only implement a *selection* of tags from the original Struts tag libraries. The tags not implemented are assumed to have counterparts in JSTL.

■ **Note** In order to use the JSTL and Struts-EL tags, you have to have a servlet container that implements the JSP 2.0 specs, which necessitates EL support. Tomcat 5.*x* is JSP 2.0 conformant.

Lab 10a: The LILLDEP Full Listing Page

In this lab, you will create a page that lists all the entries in the LILLDEP database.

Step 1: Complete ListingAction

Complete the implementation of the `Action` subclass `ListingAction` so that it prepares an `Iterator` over the LILLDEP database. The following line of code shows how to get an `Iterator` containing all `Contacts` in the database:

```
Iterator everything  = ContactPeer.doSelect(new Criteria());
```

(Refer to the sections on Lisptorq in Appendix A for details.) Ensure that your `ListingAction` implementation does the following:

- Forwards to the page named `success`.
- Puts the `Iterator` of `Contacts` on the `request` object (refer to the earlier section on `<logic:iterator>` to see how this is done). Instead of using an ad hoc name for the object, use a suitable name on the interface `JSPConstants`. This is helpful because all names shared between JSP and `Action` classes are stored in one file, `JSPConstants`.

Step 2: Complete listing.jsp

Complete the implementation of `listing.jsp`, using `<logic:iterate>` to list the following fields in a table:

- `name`
- `email`
- `company`
- `address`
- `postcode`
- `website`

When users click on `website`, it should take them to the company's website.

Step 3: Amend web.xml

Put in a declaration in `web.xml` for the Struts Logic tag library.

Step 4: Amend struts-config.xml

Put in a new action mapping in `struts-config.xml`:

- The path should be `/Listing`.
- This action mapping should tie together `ListingAction` and `listing.jsp`.

Question: Will you need a form bean for this action mapping? (See Appendix D for the answer.)

Compile, redeploy, and test your work. The main `full.jsp` page has a link on the toolbar called Full Listing that invokes the handler `Listing.do`.

■ **Note** Remember that you will have to reenter contacts because redeployment destroys the old database.

Lab 10b: Simplifying ContactForm

In the earlier section on the nested tag library, I mentioned that you can use the Nested tag library or nested properties to simplify `ActionForm` subclasses.

This is certainly the case for `ContactForm`, which has many getters and setters. Most of these can be removed if you use the nested library or nested properties.

Complete the following steps.

Step 1: Amend ContactForm

Remove all getters and setters on `ContactForm` except `getModel()` and `setModel()`. Question: If you compiled and deployed LILLDEP now, would it work? Why or why not? (See Appendix D for the answer.)

Step 2: Amend full.jsp and mnc.jsp

Use either the Nested tag library or nested properties to ensure that all fields and error messages display correctly on both these forms.

Question: In either approach, what is the new base object? (See Appendix D for the answer.)

■ **Note** If you choose to use the Nested tag library, remember to add a section in `web.xml` to declare it.

Compile, redeploy, and test your work.

Lab 10c: Using JSTL

In this lab, I'd like you to use JSTL's `<c:forEach>` tag rather than the `<logic:iterate>` tag in `listing.jsp`.

Step 1: Install the JSTL and Struts EL Tag Libraries

Open the Struts distribution found in the Source Code section of the Apress website at `http://www.apress.com`; it's contained in a zip file named `jakarta-struts-1.2.6.zip`.

1. Copy all the JAR files in the `.\contrib\struts-el\lib\` folder to LILLDEP's `\lib` folder, taking care to *overwrite* the existing files.

2. Copy all TLD files to LILLDEP's `\web\WEB-INF\` folder. Again, overwrite the older files.

Step 2: Amend web.xml

Add a `<taglib>` entry declaring the JSTL core tag library, `c.tld`. The recommended URI is `http://java.sun.com/jstl/core`, but of course, you're free to use anything you like.

Step 3: Amend listing.jsp

In `listing.jsp`, use JSTL's `<c:forEach>` tag rather than the `<logic:iterate>` tag. Remember to add to add a taglib declaration for the JSTL core tag library at the start of the page.

As usual, compile, redeploy, and test your work.

Useful Links

- Struts Apache online reference for the Logic tag library: `http://struts.apache.org/struts-doc-1.2.x/userGuide/dev_logic.html`
- Struts Apache online reference for the Nested tag library: `http://struts.apache.org/struts-doc-1.2.x/userGuide/dev_nested.html`
- *Pro JSP 2, 4th Edition*, by Simon Brown, Sam Dalton, Daniel Jepp, Dave Johnson, Sing Li, and Matt Raible (Apress, 2005)
- JSTL specs: `www.jcp.org/en/jsr/detail?id=052`

- An EL tutorial from Sun: `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html`
- A list of Struts tags that have not been EL-enabled because they have equivalents in JSTL: `http://struts.apache.org/struts-doc-1.2.x/faqs/struts-el.html`

Summary

- The Logic library has tags for iteration, conditional processing, and flow control. Most of these tags have JSTL equivalents.
- The Nested tag library is used to shift the implicit base object that the `property` attribute refers to. You can use nested properties to achieve a similar effect.
- JSTL consists of four tag libraries (Core, Formatting, XML Processing, and Database Access), which you can use in place of some Struts tags.
- `<c:forEach>`, `<c:if>`, `<c:choose>...<c:when>`, and `<c:out>` tags from the JSTL core tag library can be used to replace the Struts `<logic:iterate>` and conditional processing tags in the Logic tag library.
- `<c:out>` can be used to replace `<bean:write>`, but supporting tags from the JSTL Formatting tag library are needed to fully replace `<bean:write>`.
- EL is a way to give EL-enabled tags dynamic attributes. EL has several special keywords to help it do this.
- There are EL-enabled Struts tags you can use alongside the original Struts tags.

CHAPTER 11

Uploading Files

In this chapter, I'll discuss a few techniques you can use to *upload* files to a server. If you've ever tried to create this functionality using JSP and servlets, you'll know that it isn't easy.

Struts gives you a really painless way to upload files. But before I show you how, it is instructive to understand a little bit of what goes on behind the scenes when a user attempts to upload a file.

The starting point for any webapp that allows file uploads is the HTML element for uploading a file:

```
<input type="file">
```

In most graphical browsers, this gets rendered as a text input field followed by a Browse button, as depicted in Figure 11-1.

Figure 11-1. *The input field for file uploading as displayed in Mozilla*

Users can either type the filename directly in the given field, or click the Browse button, which launches a dialog box allowing the user to visually navigate to the desired file.

When the user clicks the submit button of the enclosing form, the web browser reads the file and encodes each byte with a suitable encoding, like uuencode (see "Useful Links"), and sends this to the server, along with any posted form data. Depending on its length, the file might also be broken into pieces.

So, seen from the server end the uploaded data is encoded and broken into several parts. To reconstruct the original file, you'd have to correctly parse the input data, rejoining broken parts, and finally decode everything. Ouch!

Struts makes things much, much easier. First, on the JSP page, you put in an `<html:file>` tag:

```
<html:file property="myFile"/>
```

As usual, the property attribute corresponds to a property on your ActionForm subclass that accepts the uploaded file. In the previous snippet, that property name is myFile.

The second change you make is to add a special property on the enclosing <html:form>:

```
<html:form enctype="multipart/form-data" action=...
```

This new property tells Struts that the data submitted by the form will be encoded and in parts.

Next, on your ActionForm subclass, you'll have to declare a property to accept the file data, as shown in Listing 11-1.

Listing 11-1. *Snippet of MyActionForm*

```
import org.apache.struts.upload.FormFile;

public class MyActionForm extends ActionForm{

   protected FormFile _file;

   public FormFile getMyFile(){
      return _file;
   }

   public void setMyFile(FormFile f){
      _file = f;
   }
     ... //rest of MyActionForm
```

In Listing 11-1, FormFile is the Struts class that handles access to the downloaded file. The fully qualified class name is

```
org.apache.struts.upload.FormFile
```

FormFile has two important functions:

- getInputStream(): Returns an InputStream to the downloaded file. You use this to read the file itself.
- destroy(): Deletes the downloaded file. The downloaded file is saved in a temporary directory, so be sure to call destroy() to free up disk space.

FormFile has a few other functions; these are listed in Appendix B.

Uploading a Fixed Number of Files at Once

From our discussion so far, it should be clear that since the uploaded file is just a property of the `ActionForm`, you can upload more than one file on a single form. If the files are to be treated equally, the best way to do this is to use indexed properties, as shown in Listing 11-2.

Listing 11-2. *A Simple Form to Upload a Fixed Number of Files*

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html>
<body>

<html:form enctype="multipart/form-data" action="/FileUpload.do">

   <h1>Upload Four Files</h1>
   File 1:<html:file property="file[0]" size="20"/><br>
   File 2:<html:file property="file[1]" size="20"/><br>
   File 3:<html:file property="file[2]" size="20"/><br>
   File 4:<html:file property="file[3]" size="20"/>
   <html:submit/><br>

</html:form>

</body>
</html>
```

The `ActionForm` associated with Listing 11-2 is described in Listing 11-3.

Listing 11-3. *ActionForm Associated with Listing 11-2*

```
import org.apache.struts.upload.FormFile;

public class FileUploadForm extends ActionForm{

   protected FormFile[] _files;

   public FileUploadForm(){
      _files = new FormFile[4];
   }

   public FormFile getFile(int i){
      return _files[i];
   }
```

```
   public void setFile(int i, FormFile f){
      _files[i] = f;
   }
     ... //rest of  FileUploadForm
```

One important thing to note from Listing 11-3 is the order of the arguments in `setFile()`. A common mistake is to reverse the order:

```
setFile(FormFile f , int i)  //WRONG!! Arguments reversed.
```

Apart from this, Listings 11-2 and 11-3 should hold no surprises.

It is possible to improve this technique somewhat if the order in which the files are specified for uploading is unimportant. Listing 11-4 is a variation of Listing 11-3's `FileUploadForm`.

Listing 11-4. *A Variation on Listing 11-3*

```
import java.util.*;
import org.apache.struts.action.*;
import org.apache.struts.upload.*;

public class FileUploadForm extends ActionForm {

   protected List _files;

   public FileUploadForm(){
      _files = new ArrayList();
   }

   public FormFile getFile(int i){
      return(_files.size() > i)? (FormFile)_files.get(i) : null;
   }

   public void setFile(int i, FormFile f){
      if(f.getFileSize() <= 0){
        f.destroy();
      }else{
        _files.add(f);
      }
   }
}
```

The advantage of Listing 11-4 is that you don't have to hard-code the number of files to be uploaded into the `ActionForm` subclass. Notice also that `setFile()` only accepts a file of nonzero length.

The downside of this approach is that there's no guarantee Struts will call the `setFile()` function in either ascending or descending order of the index variable. This means that subsequent retrievals of the uploaded files might not be in the order specified by the user.

The main drawback with both these simple techniques is that you have to know beforehand the number of files to be uploaded. Even with the second technique, the number of `<html:file>` tags had to be hard-coded in the JSP. There's no way for a user to upload an arbitrary number of files. In some applications (e.g., a webmail app), this might be too restrictive. I'll show you how to overcome this shortcoming next.

RESTRICTING THE SIZE OF UPLOADED FILES

In Chapter 9, I described how you can limit the maximum allowable file size to accept in a single upload. You use the `<controller>` tag of `struts-config.xml` to do this. For example:

```
<controller maxFileSize="2M" />
```

sets a limit of 2MB. You use a numeric value followed by `K`, `M`, or `G` to indicate a size in kilobytes, megabytes, or gigabytes.

Uploading Any Number of Files

Struts does not provide a one-stop solution for uploading an arbitrary number of files. However, it does provide adequate tools for you to "roll your own" solution. In this section, I'll describe one solution to this interesting problem. (For expositional clarity, the solution I'll describe will *not* preserve the order of the uploaded files, but it isn't too difficult to fix this.)

The heart of the solution is to have *two* submit buttons for the file upload form:

- The first submit button is an Upload More Files button (or something equivalent, like a link). This button simply forces Struts to save the uploaded files, and then redisplays the input page with the list of files uploaded thus far. The values keyed in on other fields are also displayed. Because this round-trip may take time, it is usually advisable to give the user more than one input field to upload files.

- The second button is the true submit button, which sends all the form data (including all uploaded files) for processing.

In what follows, I'll illustrate the solution in the context of a trivial webmail app. This webapp has just two pages: the first allows the user to compose an email message, and the second page displays the contents of the email. Figure 11-2 shows what the compose page looks like, while Figure 11-3 depicts the output page.

Figure 11-2. *The compose page*

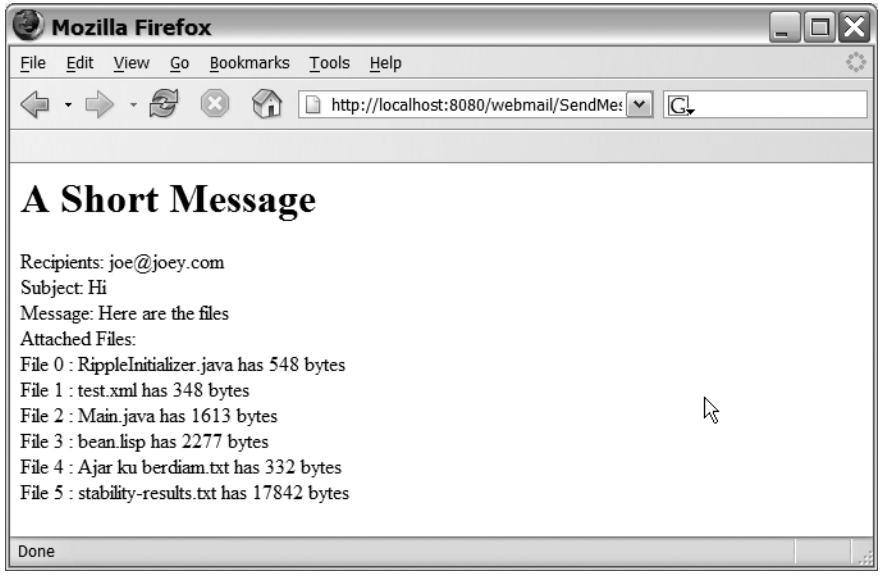
Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/webmail/SendMes
A Short Message
Recipients: joe@joey.com
Subject: Hi
Message: Here are the files
Attached Files:
File 0 : RippleInitializer.java has 548 bytes
File 1 : test.xml has 348 bytes
File 2 : Main.java has 1613 bytes
File 3 : bean.lisp has 2277 bytes
File 4 : Ajar ku berdiam.txt has 332 bytes
File 5 : stability-results.txt has 17842 bytes
Done

Figure 11-3. *The output page (showing six attached files)*

Struts has a robust way for you to put two submit buttons on a single form. I've described this technique in detail in Chapter 17's `LookupDispatchAction`. Please read that section before proceeding. In what follows, I'll assume that you know how `LookupDispatchAction` works.

The starting point of the solution is Listing 11-4's `FileUploadForm`. This `ActionForm` subclass contains all the functionality we need to upload an arbitrary number of forms. In order to accommodate the other fields of the email, we subclass `FileUploadForm`, as in Listing 11-5. This is good design since we can reuse the file uploading functionality in other forms.

Listing 11-5. *WebmailForm.java*

```
package net.thinksquared.webmail;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public class WebmailForm extends FileUploadForm{

   private String _recipients;
   private String _subject;
   private String _message;

   public String getRecipients(){
      return _recipients;
   }

   public String getSubject(){
      return _subject;
   }

   public String getMessage(){
      return _message;
   }

   public void setRecipients(String recipients){
      _recipients = recipients;
   }

   public void setSubject(String subject){
      _subject = subject;
   }
```

```
    public void setMessage(String message){
        _message = message;
    }

    public ActionErrors validate(ActionMapping mapping,
                                 HttpServletRequest request){

        ActionErrors errors = new ActionErrors();

        //NO BLANK MESSAGES/RECIPIENTS/SUBJECTS
        if(_recipients.trim().length() == 0){
          errors.add("recipients",
                     new ActionMessage("webmail.error.recipients"));
        }
        if(_subject.trim().length() == 0){
          errors.add("subject",
                     new ActionMessage("webmail.error.subject"));
        }
        if(_message.trim().length() == 0){
          errors.add("message",
                     new ActionMessage("webmail.error.message"));
        }

        return errors;
    }

}
```

Listing 11-6 contains the JSP page that displays the compose page shown in Figure 11-2. For clarity, and to highlight the use of message keys in conjunction with `LookupDispatchAction`, I've not used message resources consistently in Listing 11-6.

Listing 11-6. *JSP for the Compose Page*

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<html>
<body>

<html:form enctype="multipart/form-data" action="/SendMessage.do">

    Recipients: <html:text property="recipients"/>
    <html:errors property="recipients"/><br>
```

```
   Subject:  <html:text property="subject"/>
   <html:errors property="subject"/><br>

   Message: <html:text property="message"/>
   <html:errors property="message"/><br>

   <h1>Attach Files</h1>
   File 1:<html:file property="file[0]" size="20"/><br>
   File 2:<html:file property="file[1]" size="20"/><br>
   File 3:<html:file property="file[2]" size="20"/><br>
   File 4:<html:file property="file[3]" size="20"/>

   <html:submit property="action">
      <bean:message key="webmail.prompt.more-files"/>
   </html:submit><br>

   <html:submit property="action">
      <bean:message key="webmail.prompt.send"/>
   </html:submit>

</html:form>

</body>
</html>
```

In Listing 11-6, note the two submit buttons, as well as the required setup for using `LookupDispatchAction` (see Chapter 17). The `Action` subclass to handle both submits appears in Listing 11-7. Notice that we do *not* override `execute()` in `SendMessageAction` because of how `LookupDispatchAction` works.

Listing 11-7. *SendMessageAction*

```
package net.thinksquared.webmail;

import java.util.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import org.apache.struts.actions.*;

public class SendMessageAction extends LookupDispatchAction{
```

```
  protected Map getKeyMethodMap(){
      Map m = new HashMap();
      m.put("webmail.prompt.more-files","attach");
      m.put("webmail.prompt.send","send");
          return m;
  }

  public ActionForward attach(ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response){

     return mapping.getInputForward();

  }

  public ActionForward send(ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response){

     return mapping.findForward("success");

  }

} buttons
```

The `struts-config.xml` file appears in Listing 11-8, and the JSP for the output page shown in Figure 11-3 is provided in Listing 11-9. Notice that the form bean is in session scope. This means that you have to perform some cleanup (for clarity, not shown in the listings here) in order to reset the form fields.

Listing 11-8. *struts-config.xml*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
    //...omitted for clarity

<struts-config>
```

```
    <form-beans>
        <form-bean name="msgForm"
                   type="net.thinksquared.webmail.WebmailForm"/>
    </form-beans>

    <action-mappings>

        <action path="/SendMessage"
                type="net.thinksquared.webmail.SendMessageAction"
                name="msgForm"
                validate="true"
                scope="session"
                parameter="action"
                input="/index.jsp">

            <forward name="success" path="/out.jsp"/>

        </action>

    </action-mappings>

    <message-resources parameter="Application"/>

</struts-config>
```

Listing 11-9. *The JSP for the Output Page*

```
<%@ taglib uri="/tags/struts-nested" prefix="nested" %>
<%@ taglib uri="/tags/struts-logic"  prefix="logic" %>
<%@ taglib uri="/tags/struts-bean"  prefix="bean" %>
<html>
<body>

<h1>A Short Message</h1>

<nested:root name="msgForm">

    Recipients:
    <nested:write property="recipients"/><br>

    Subject:
    <nested:write property="subject"/><br>
```

```
    Message:
    <nested:write property="message"/><br>

</nested:root>

    Attached Files:<br>

    <logic:iterate name="msgForm" property="files" id="file"
                indexId="i">

      File <bean:write name="i"/> :
      <bean:write name="file" property="fileName"/>
      has <bean:write name="file" property="fileSize"/> bytes<br>

    </logic:iterate>

</body>
</html>
```

You'll find the source code for Listings 11-4 to 11-9 in the Source Code section of the Apress website, found at `http://www.apress.com`, in the file named `multiple-file-uploading.zip`. I encourage you to copy this file to your development directory, unzip it, and run `compile.bat`. Deploy the WAR file and play with the webapp until you are confident that you understand the material presented in this section.

When you are done, implement a solution that gives the uploaded files in the order the user uploaded them. When you have finished implementing, deploying, and testing your solution, proceed to Lab 11.

Lab 11: Importing Data into LILLDEP

In this lab session, you'll create a handler that allows users to upload `Contacts` into LILLDEP. The uploaded data has to be in comma-separated values (CSV) format, which should be familiar if you've ever used Microsoft Excel:

```
Name | Email | Department | ...
Joe | joe@joey.com | ITC Department | ...
...
```

In the example, the separator is a pipe symbol (`|`), not the usual comma. The first row gives the column names, and subsequent rows are data.

Complete the following steps to provide this new "importing" facility for LILLDEP.

Step 1: Complete ImportForm

Complete the code for `ImportForm`. Specifically, put in `getFile()` and `setFile()` functions. Do you need to implement the `validate()` or `reset()` method?

Step 2: Complete import.jsp

1. Put in a form to upload a CSV file. The form handler should be `ImportForm.do`.
2. Add an `<html:file>` tag and appropriate labels. What should the `property` attribute of `<html:file>` be? Remember to use only messages from the `Application.properties` file.
3. Be sure to put in an `<html:errors>` tag to display error messages in uploading or processing the file.

Step 3: Complete ImportAction

There is a LILLDEP utility class called `CSVIterator` that can read and parse a CSV file. The constructor accepts a `Reader` for data input and a `String` for the separator (for this lab, you'd use the pipe symbol for a separator):

```
public CSVIterator(Reader input, String separator)
```

This class is an `Iterator` and returns a `Map` when `next()` is called. This `Map` contains entries corresponding to each column in the CSV file. The idea is to create a `Contact` from this `Map`, and then save the `Contact`. You can easily do this with one of the `Contact` constructors:

```
public Contact(Map data)
```

With this information, complete the `ImportAction` so that it uses the uploaded file to populate the LILLDEP database. (Hint: You can "convert" an `InputStream` to a `Reader` using `InputStreamReader`.)

In your implementation, be sure to

- Display any uploading/processing errors to the user by redisplaying the import page with errors.
- Forward to `success` if everything is OK.

Step 4: Amend struts-config.xml

Complete the following:

1. Declare a new form bean for `ImportForm`.
2. Define a new form handler to handle the importing. What is the `path` attribute of the form handler?
3. Put in a forward called `success` to point to `/Listing.do`.

Step 5: Compile, Redeploy, and Test Your Application

The `full.jsp` page has a link on the toolbar area called `Import`. Use the file called `.\csv-data\test.csv` to test your import facility. The file should upload and process without errors, and you should see the uploaded entries in the "Full Listing" page you created in Lab 10.

Useful Links

- Uuencode entry in Wikipedia: `http://en.wikipedia.org/wiki/Uuencode`

Summary

- The `<html:file>` tag is used to upload files to the server.
- The `FormFile` class handles access to the downloaded file.

CHAPTER 12

Internationalization

Simply put, internationalization (*i18n* for short, since there are 18 letters between the *i* and *n* of *internationalization*), or localization, is the job of getting an application to work for languages other than the one it currently targets.

Localizing a web application is by no means a simple task. There are subtleties beyond merely translating web pages to a new language. To understand these issues, consider the four areas that need to be addressed when you localize an application:

- **Processing input**: Since the input data comes from a web browser, your webapp has to be able to correctly handle different *character encodings* (more about this in the next section).
- **Validations**: Users in the new language group may use different conventions for dates or numbers. Your validations will have to correctly handle these.
- **Output**: Dialog boxes, prompts, button labels, and other text have to be translated to the new language. Ideally, users should be able to choose the language with which they can interact with the webapp. Struts solves these problems particularly well.
- **Data storage and related issues**: You might have to ensure that your database can handle multibyte character encodings. A similar issue crops up when your web application passes data to other applications—you have to ensure that these applications accept text in the new language and character encoding.

In this chapter, you'll see how Struts greatly simplifies the task of localizing a webapp by addressing the first three areas. But first, I'll have to define a few terms.

Character Encodings, Unicode, and UTF-8

A **character** is an abstract entity, corresponding to a single written unit in a language or script. For example, the letters "A" or "A" or "A" all embody the same, single character.

"A" and "a" represent different characters, though, because they carry different meanings. Characters also include punctuation marks and other symbols in a script.

A **character set** is a collection of characters along with a unique numeric identifier for each character. In other words, a character set is just a collection of (character, identifier) pairs. The characters in a character set may come from one or more languages or scripts.

Unicode is a character set that aspires to encompass characters from all human languages and scripts.

■ **Note** Strange as it may sound, a single language may have more than one script. For example, the Uyghur language (from central Asia) has three scripts: the old Arabic script, the Cyrillic-like script used in ex-Soviet countries, and a Latin script invented by the Chinese government and now used by Uyghurs to communicate over the Internet.

Unicode (currently in version 4.0) is defined by the Unicode Consortium, an organization backed by many (mainly U.S.) companies. This consortium has identified every single character in every (well, almost every) language on the planet and assigned to each character a unique number. The numbering is done so that each language gets one or more contiguous block of numbers.

A **character encoding** transforms each numeric identifier in a character set into another number called a **character code**. This is done because the character code is more convenient or efficient for storage on computers. So, a character encoding is simply a collection of (character, character code) pairs.

Now, if *no* transformation needs to be done—the numeric identifiers of the character set are used unchanged—then of course the character encoding and the character set are the same. An example of this is ASCII, which defines both a character encoding and a character set. Other examples are Latin 1 and ISO encodings for various languages.

Back to Unicode: the downside is that it is practically useless as a *character encoding* for everyday programming since the existing character encodings, notably ASCII, are firmly entrenched. Many existing programs or software libraries accept only ASCII input and will not work with the Unicode numeric identifiers.

UTF-8 is a *character encoding* for Unicode that solves this problem. UTF-8 takes each Unicode-defined numeric identifier and transforms it into a character code that may occupy one or more bytes of storage. The brilliant thing about UTF-8 is that characters belonging to both Unicode and ASCII are given the same character code.

This means that your "UTF-8 enabled" program can take advantage of Unicode and still work with legacy programs or libraries that accept only ASCII. Of course, this doesn't solve the problem of getting legacy programs to accept Unicode. The point is that you *don't completely* lose the ability to work with legacy software.

This brings us to two questions. If the numeric identifiers defined by Unicode aren't actually used by programs, what good is Unicode? And why the need to separate Unicode from UTF-8?

To understand the advantages of Unicode, you have to see that it's a *widely accepted standard* for labeling *all* characters with unique numeric identifiers. So, if a web browser and a server agree that textual data between them represents Unicode characters, you could develop webapps that handle *any* language. If there were no widely accepted standard for all characters, then this would not be possible. Before Unicode, character sets were either not widely accepted or did not include all characters.

To see why Unicode has to be separate from UTF-8, you must understand the different goals of each. The Unicode numeric identifiers represent a character and hence have *meaning*. The UTF-8 character encoding is about getting existing programs to work. These are different goals and best solved in different ways.

For example, not all legacy software uses ASCII (think ex-Soviet Union), so hard-coding the UTF-8 character encodings into Unicode (that is, making the Unicode numeric identifier equal the UTF-8 character encoding) would not be beneficial to everyone. Also, other encodings of Unicode are more efficient in terms of storage space for non-Latin languages. Hard-coding UTF-8 into Unicode would force users of these languages (think China) to use more storage space than necessary. Both these factors would hinder the widespread adoption of Unicode.

So, the separation between Unicode and any encoding of it (like UTF-8) gives users the benefit of being able to agree on the meaning of data *and* the flexibility of choosing a character encoding that best meets their needs.

WHAT ABOUT PLAIN TEXT?

If by "plain text" you mean textual data without any encoding, I hope from the preceding discussion you realize that there's no such thing as "plain text"!

Any textual data has an implicit character encoding. In the Western world, this might be ASCII.

The important thing to understand is that you can either *assume* an encoding—most programs and software assume an encoding, like ASCII—or you can make an encoding explicit, as is the case in the headers of XML or HTML documents. However, ensuring that the *declared* encoding matches the *actual* encoding (done by the text editor) is a separate issue.

Lastly, just as it's possible for the binary value of a Java integer to be displayed as a string, it's possible for the numeric identifier of a Unicode character to be displayed as a string. These are usually displayed in the ASCII encoding as a hexadecimal number, like `\u7528`. This partially circumvents the problem of "using" Unicode in legacy software. You'll learn more about this in a later section ("Localizing Output"). *Do not* confuse this ASCII representation of the Unicode numeric identifier with UTF-8.

Locales

In the previous section, I described how using Unicode and the UTF-8 encoding would allow client-server programs to communicate data in any language.

However, this isn't quite enough. What's needed is a mechanism for users to *select* the language they want. For example, if you are a Chinese speaker, you might want to ask my web application to be presented to you in Chinese.

Locales are a widely used mechanism to achieve this. A locale is an identifier for a language or a variant of a language. For example, the identifier for U.S. English is `en_US` and the identifier for Swahili is `sw`.

The local strings themselves are a combination of language code and possibly country code. The language code is a two-letter string from the ISO 639 standard. The country code (e.g., the `US` in `en_US`) is taken from the ISO 3166 standard. Notice that the two-letter language code is in lowercase and the two-letter country code is in uppercase, joined by an underscore.

Essentially, the web browser specifies the desired locale as an extra parameter in the HTTP header. The server responds by sending that user all pages in that locale, if possible.

Locales are the primary basis for localization in Struts. All of the internationalization features of Struts are geared toward easy customization of your application by locale. You'll see this in action in this as well as future chapters.

Now that you understand the basics of character encodings, Unicode, UTF-8, and locales, it's time to move on to address the three primary areas of concern you'll encounter when you attempt to localize your Struts web application.

Processing Input

Web browsers might use a variety of character encodings, depending on the user's default locale or preferences. Since the input data you're going to work with comes from a web browser, your webapp potentially has to handle arbitrary character encodings.

Now, the last thing you want to do is accept and process different character encodings. And why not? Suppose you store text from different encodings in your database. How will you ever know later which encoding they come from? You'd have to store the encoding information along with the each textual data you save. Not pretty.

The obvious solution is to use a single encoding that handles *all* languages—like UTF-8.

However, here you run into a problem: there's *no way* to fix the character encoding you receive from the user's web browser. Browsers will submit form data in any old character encoding and there's just no way around this.

■ **Note** With the HTML 4.0 specification, web servers can set the `accept-charset` parameter in the HTTP response, but most browsers will ignore this.

One reasonable assumption you can make is that browsers will respond with the same encoding they receive from the server. So, if your HTML form is in UTF-8, it's a safe bet that the responses will be in UTF-8 too. This isn't always true, since web browsers offer users the option to change the default encoding.

So, here's a good rule to stick to. *Use a single, universal character encoding (like UTF-8) consistently throughout your application, in every JSP, HTML, XML page, and so forth.*

For your JSPs you could use the `<%@ page` declaration:

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

or you could use the `<controller>` tag in your `struts-config.xml` (see Chapter 9):

```
<controller contentType="text/html; charset=UTF-8" />
```

This second method means you need not embed the `<%@ page` declaration in all your JSPs. Of course, this trick wouldn't work if you call the JSP bypassing Struts. If you're calling the JSP by its name (e.g., `myPage.jsp`), you're bypassing Struts. If the page is delivered from a `<forward>` or a `.do` extension, then you are not bypassing Struts, and the trick will work.

For your static HTML pages, you could use the `<meta>` tag to specify the encoding:

```
<head>
   <meta http-equiv="content-type" content="text/html; charset=UTF-8">
</head>
```

You have to exercise caution here, because you must ensure that the actual encoding of the HTML file (determined by your text editor that writes the HTML) is the same as the declared encoding (declared in the `<meta>` tag). There's no way to tell what encoding a browser will respond with if given a form that says it's in UTF-8 but that is really in another encoding.

Localizing Validations

By "localizing validations," I mean performing validations specific to a locale. For example, if a German user (locale `de`) keyed in a date, you might want to validate the input according to the format `dd.mm.yyyy`. For a U.S.-based user, you might use a different format, probably `mm/dd/yyyy`.

There is currently very poor support for doing this with Struts. In Chapter 15, when I cover the Validator framework, you'll see the *only* support Struts has for localizing validations.

To be fair, there aren't many situations where localizing validations would be useful. In fact, in many situations, it would be wrong! For example, there are users who use an `en_US` locale but don't actually live in the United States. So you can't reliably validate things that depend on geography like postal codes (zip codes in the United States) by locale.

However, for things like dates, currency values, or numbers, localizing validations *may* be necessary. This section contains my personal collection of hacks that I hope you will find useful.

The first hack is to test for each special locale in your `ActionForm` subclass (see Listing 12-1).

Listing 12-1. *Localizing Validations by Brute Force*

```
String locale = getLocale();
String date = getDate();
boolean dateOK = false;

if ("de".equals(locale)) {
   dateOK  = validateDate(date,"dd.mm.yyyy");

}else if("en-us".equals(locale)){
   dateOK  = validateDate(date,"mm/dd/yyyy");

}else{ //catch all date validation
   dateOK  = validateDate(date,"dd/mm/yyyy");
}
```

In Listing 12-1, `getLocale()` is a function on the `ActionForm` base class. `validateDate()` is a fictitious function that validates a date string given a second format string. This brute-force method is OK for doing a few validations by locale, but it has many disadvantages, a maintenance nightmare being one of them!

A slightly more sophisticated way of doing the same thing is to use a `HashMap` to store the (`locale`,`format`) pairs. This removes the multiple `if else` statements, as shown in Listing 12-2.

Listing 12-2. *Localizing Validations by Brute Force, Take 2*

```
public MyActionForm(){

   myDateFormats = new HashMap();
   myDateFormats.put("en-US"," mm/dd/yyyy");
   myDateFormats.put("de","dd.mm.yyyy");
   myDateFormats.put("default","dd/mm/yyyy");
}

public ActionErrors validate(...){

   ...
```

```
    boolean  dateOK = false;

    Object o =  myDateFormats.get(getLocale());
    if(null == o) o = myDateFormats.get("default");

    dateOK  = validateDate(getDate(), (String) o);

    ...

}
```

The second hack is a clever trick that appeared in an article by Mike Gavaghan in JavaWorld (see "Useful Links" at the end of this chapter). The idea is to embed the format string within the JSP, using the `<html:hidden>` tag, as shown in Listing 12-3.

Listing 12-3. *Using Hidden Format Fields (JSP)*

```
<bean:message key="myapp.jsp.prompt.date"/>
<html:text property="date"/>
<html:hidden property="dateFormat"/>
```

Before this page is called, you need to populate the hidden field using an Action, with the locale's date format. Now, since the date format is stored in the form itself, you can retrieve it and run the validation with the format, as shown in Listing 12-4.

Listing 12-4. *Using Hidden Format Fields (ActionForm)*

```
public ActionErrors validate(...){

    ...

    boolean  dateOK = false;
    dateOK  = validateDate(getDate(), getDateFormat());

    ...

}
```

Listing 12-4 implies the addition of `getDateFormat()` and `setDateFormat()` functions on `MyActionForm`. This technique allows you to store your locale-specific formats along with the properties file for that locale. This solution is better than hard-coding formats into your `ActionForm` subclass.

Localizing Output

Localizing output is something Struts does very well. If you've been careful to use `<bean:message>` tags throughout your application's JSPs (instead of using static text), then localizing your application's output with Struts is easy.

Since all your application's text is in the `Application.properties` file, here's all you need to do to support a new language:

- **Translate** the `Application.properties` file to the language for the new locale. The translation would leave the keys the same, and only affect the messages. If the translator used a character encoding other than ASCII, you will have to postprocess the translated file before you can use it. More on this shortly.

- **Append the locale identifier** to the translated filename. For example, a Japanese translation must be called `Application_jp.properties`. The additional `_jp` identifies this translation as belonging to the Japanese locale.

- **Install** the new properties file in the *same folder* as the default `Application.properties` file (see Figure 12-1).

That's it! When the user's browser is set for the desired locale (say `jp`), Struts will use the right properties file (in this case, `Application_jp.properties`) for that user. All prompts, buttons, dialog boxes, and text will appear in Japanese.

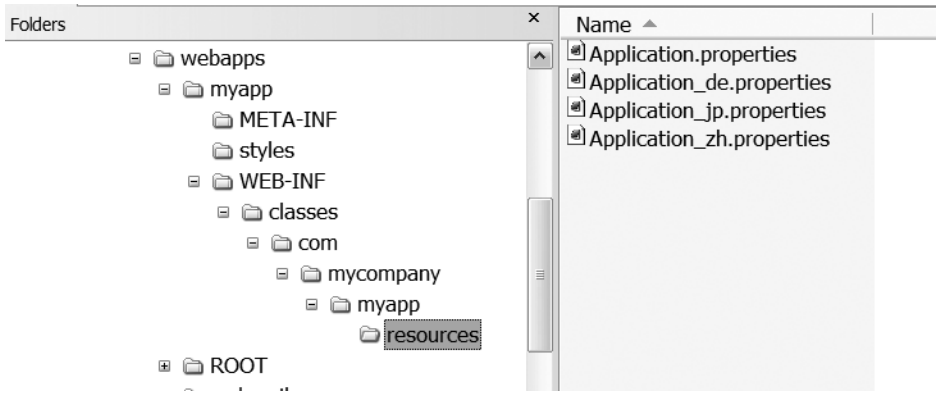

Figure 12-1. *Various installed Application.properties files*

■ **Note** If you were writing an application for U.S. English (with the associated locale `en_US`), then the properties file would be `Application_en_US.properties`.

Before I show you how this selection is made, let's tie up one loose end—the character encoding for the translated `Application.properties`.

Processing Translated Application.properties Files

The Java programming language only accepts source code and properties files in the ISO 8859-1 character encoding. I'm talking the actual source code here—not the internal representation of Java `String`s, which are not restricted to this encoding.

The ISO 8859-1 (aka Latin 1) can encode most western European languages, and is for most intents and purposes like ASCII. More precisely, characters that are in both ASCII and ISO 8859-1 are given the same character codes.

Your `Application.properties` file must also be in this encoding. How do we represent other languages with this restrictive encoding? The answer is to use a tool called `native2ascii`. This program resides in the `bin` directory of your JDK.

When you receive the translated `Application.properties` file, it might likely not be in the ASCII or ISO 8859-1 encoding. The only sure way of getting this information is from the translator. Once you know the source encoding, you can use `native2ascii`:

```
native2ascii -encoding UTF-8 jp.properties Application_jp.properties
```

In this example, the translated file is `jp.properties`, which is given in the UTF-8 encoding. The final output is `Application_jp.properties`, which you can actually use.

The "magic" behind `native2ascii` is that it "escapes" non–ISO 8859-1 characters with the string Unicode version of it. This appears in your properties file as sequences of `\uxxxx`.

When Struts creates the HTML page to be delivered to the user, the encoding specified on the page with the `<%@ page ...` directive is used:

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

or, you could use the `<controller>` tag in your `struts-config.xml` (see Chapter 9):

```
<controller contentType="text/html; charset=UTF-8" />
```

In both these cases, the UTF-8 character encoding is used to encode the HTML pages delivered to the user.

Selecting a Locale from the Browser

Most modern browsers allow the user to select a default locale. Figures 12-2 and 12-3 show this for Mozilla and Internet Explorer.

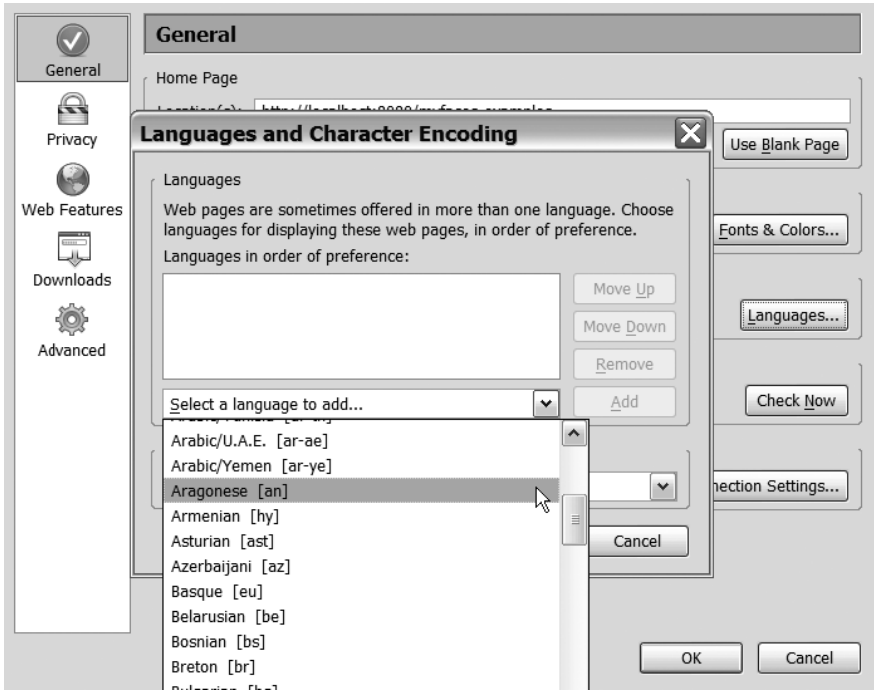

Figure 12-2. *Adding a new locale with Mozilla*

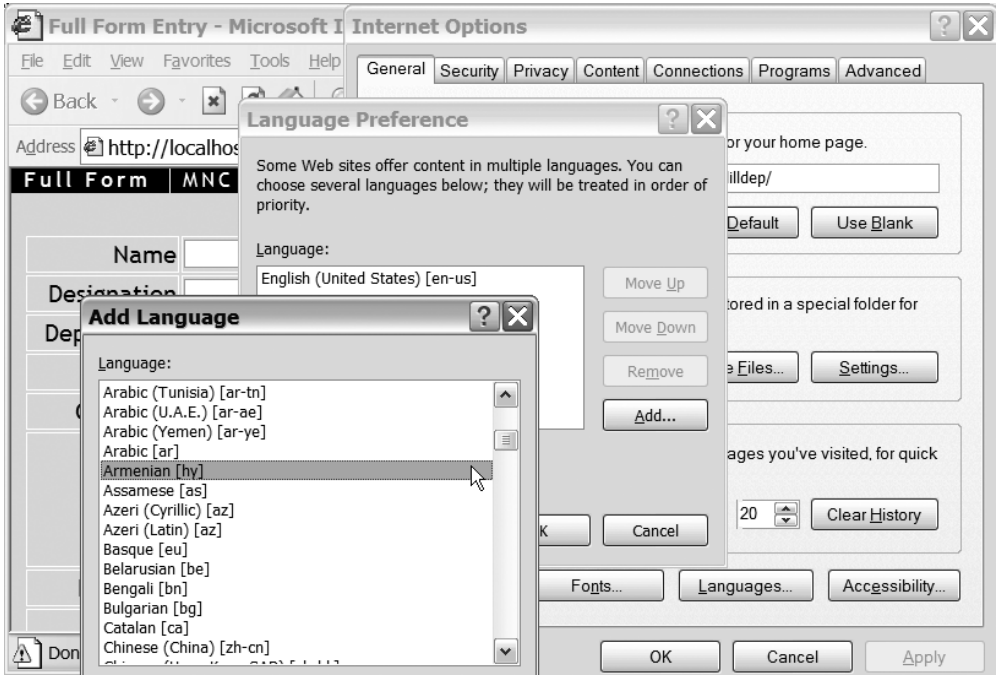

Figure 12-3. *Adding a new locale with Internet Explorer*

You may notice that these web browsers don't always give the correct locale string. For example, U.S. English is incorrectly displayed as `en-us`, when it is in fact `en_US`.

Switching Locales with a Link

Getting users to change the locale on their web browser may not always be desirable or even possible. In such situations, you might want them to click a link instead that tells the server to switch to the desired locale. The way to do this is to encode the locale as a parameter on the link. For example:

```
<a href="/ChangeLocale.do?language=ta">In Tamil</a>
```

In this link, `ChangeLocale` is a form handler, which has no associated form bean. The declaration in `struts-config.xml` would be

```
<action path="/ChangeLocale"
        type="com.mycompany.myapp.ChangeLocaleAction">

        <forward name="success" path="/index.jsp"/>
</action>
```

`ChangeLocaleAction` is an `Action` subclass with the following `execute()`:

```
public ActionForward execute(...){

   String language = request.getParameter("language");
   String country = request.getParameter("country");

   if(null == country){
      setLocale(request, new Locale(language));
   }else{
      setLocale(request, new Locale(language,country));
   }

   return new ActionForward(mapping.findForward("success"));
}
```

When the user clicks the link (which contains the new language, and optionally, the new country code), control passes to `ChangeLocaleAction`, which sets the default locale for that user's session to the desired combination. Note that `java.util.Locale` is a built-in Java class.

Switching Locales with LocaleAction

Yet another way to allow users to switch locales is to use the Struts class `LocaleAction`. To use this class *effectively*, you'll have to first understand dynamic forms (see Chapter 16), so I've postponed the discussion of `LocaleAction` to Chapter 17.

Lab 12: LILLDEP for the Malaysian Market

In this lab, you'll see how easy it is to port LILLDEP for the Malaysian (or Indonesian) market.

1. In the Source Code section of the Apress website, found at `http://www.apress.com`, copy `lab12.zip` to your main LILLDEP development folder. This zip file should contain a single file called `Application_ms.properties`; `ms` is the locale string for Malay, the language spoken in Malaysia (and a slight variant of it is spoken in Indonesia). Unzip the contents of this zip file to `.\web\resources\`.

2. Compile and redeploy LILLDEP.

3. Open Internet Explorer, then choose Tools ➤ Internet Options ➤ Languages (see Figure 12-2). If you're using Mozilla or another browser, you'll have to locate this setting yourself. Add Malay (`ms`) and remove all other languages.

4. Close all instances of Internet Explorer, then open one and navigate to the LILLDEP page. You should see all prompts and buttons in Malay (see Figure 12-4).

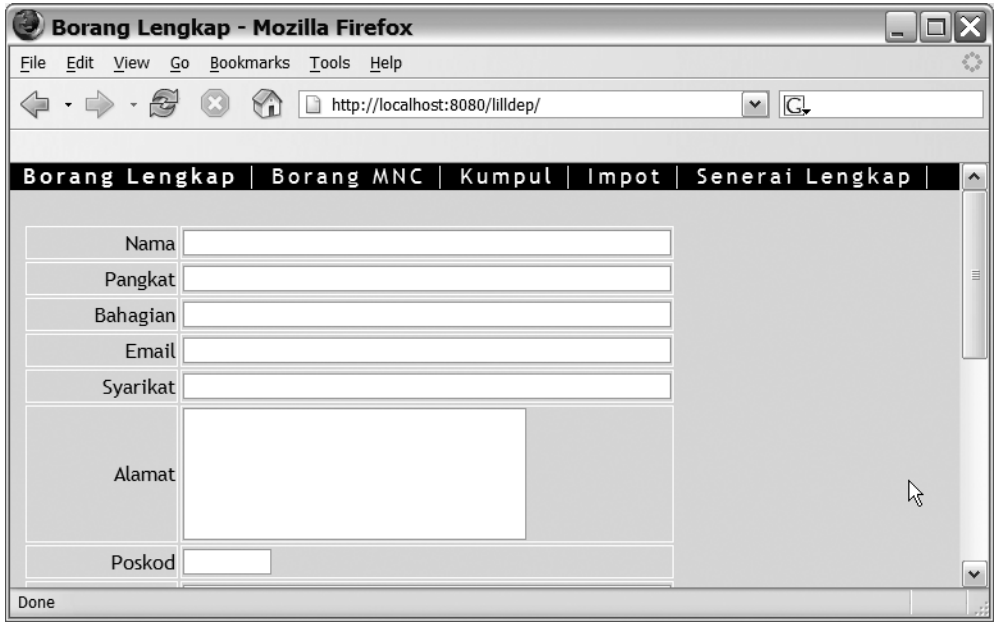

Figure 12-4. *LILLDEP in Malay*

Useful Links

- "End-to-end internationalization of Web applications: Going beyond the JDK," by Mike Gavaghan: `http://www.javaworld.com/javaworld/jw-05-2004/jw-0524-i18n.html`
- ISO 639 language codes: `http://en.wikipedia.org/wiki/ISO_639`
- ISO 3166 country codes: `http://en.wikipedia.org/wiki/ISO_3166`
- Unicode: `www.unicode.org`
- UTF-8 and related encodings: `http://en.wikipedia.org/wiki/UTF-8`

Summary

- Localizing occurs in four areas: input, validations, output, and communication with other programs, like databases.
- Struts provides support for localizing output but poor or nonexistent support for the other areas.
- Localizing output is as simple as getting the `Application.properties` file translated.
- The properties file needs to be encoded in ISO 8859-1 (Latin 1). You might have to use the `native2ascii` tool to "escape" other encodings into this one.

CHAPTER 13

Review Lab: Editing Contacts in LILLDEP

Congratulations! You've made it halfway through this book. Before moving on to more advanced topics, I'd like to give you the opportunity to review some of the material in the preceding chapters.

In this review lab, you'll make it possible to edit `Contacts` in LILLDEP's database. Specifically, you'll build on the "full listing" page (`listing.jsp`), so that when a user clicks on a company name, they are taken to a page displaying the `Contact`'s full details. These details may be changed and resubmitted, causing the database entry to be updated.

Try your best to answer questions 1–4:

1. If you turn the company name into a link, how will you determine which company was clicked? (*Hint*: Look at the source code for `BaseContact`.)

2. You obviously want to reuse `full.jsp` to display the data that's going to be edited. Do you need to make any changes to it to support updating contact information? Why?

3. Can you similarly reuse `ContactForm` and `ContactAction`? Do you need to make changes to them to support updating?

4. What other classes would you need to complete the editing facility? (*Hint*: What creates the populated form for editing?)

Compare your answers to the ones in Appendix D before proceeding.

Implementing the Edit Facility

All you need to do is populate and display the `full.jsp` form once the user clicks on the company name. You have already implemented the code for updating the `Contact` in the previous lab sessions. Complete the following:

1. Amend `listing.jsp` to put in the link for the company name. Let the handler be `EditContact.do`.

2. Complete the implementation of `EditContactAction` to load the form data. This `Action` should forward to `full.jsp`.

3. Put in an action mapping to tie the path `EditContact` with `EditContactAction` and the form `ContactForm`.

Test out your application to see if the form populates correctly. Also ensure that you can make changes to the contact's details.

PART 2

Advanced Struts

As Struts became a common starting point for developers new to the Java platform, an interesting phenomenon was occurring—for many developers, the key perceived value of using Struts was assumed to be the JSP custom tags for HTML forms. While these tags are quite useful, they do not constitute a robust user interface component model, which has led to the need to create or integrate third-party tag libraries for more complex presentation requirements. To me, the core value has always been in the controller tier—the request processing lifecycle, and the features which this lifecycle has enabled, such as the additions of the Tiles framework for reusable look and feel, and the Validator Framework for client-side and server-side enforcement of form validation business rules.

—Craig McClanahan

CHAPTER 14

Tiles

Tiles is a Struts *plug-in*—a piece of software developed independently of Struts, to extend the basic capabilities of Struts.

Tiles was developed initially by Cedric Dumoulin but is now (since version 1.2) integrated into Struts. In fact, it supercedes an older "Template" library, which had provided limited layout capability. The only thing that betrays its previous independent existence is that it's a plug-in and therefore requires some setup.

Tiles gives Struts applications two new capabilities:

- **Layouts**: The ability to easily provide a uniform "look and feel" to your Struts applications. In most web applications, pages follow a similar *layout*—for example, they might have a common header or footer or sidebars. Tiles has a elegant inheritance mechanism that allows such layouts, even complicated ones, to be created and maintained easily.

- **Components**: The ability to build reusable GUI components that you can easily embed into your JSP pages. Components give you the flexibility to display *dynamic content*—content that can change depending on user interaction or on external data sources. An example is displaying a list of weather forecasts for various cities, or a news items that interest the user, or targeted marketing.

In short, layouts provide a common look and feel, and components allow provision of dynamic content in a reusable package.

The powerful thing about Tiles is that you can mix these two capabilities: your layouts may include components. For example, you can specify a layout that displays a component as a sidebar containing a user's search list. Used wisely, this can lead to easy-to-use applications. You'll create such a component yourself in the lab section of this chapter.

Of course, both layouts and component behavior can be emulated in other ways. For example, you could enforce a common look and feel by simply pasting the appropriate HTML in your JSP pages. You could implement "components" by cutting and pasting scriptlets in your JSPs!

If you've followed the discussion on MVC in Chapter 5 closely, the shortcomings of these approaches should be obvious. MVC is all about separating code according to their functionality, and violating this principle leads to maintenance woes down the road. So it is with these ad hoc approaches. With them, making site-wide changes to layouts or components are difficult because you'd have to amend each page.

As you will see, Tiles does it better—and easier.

Installing Tiles

To use Tiles, you have to declare it in `struts-config.xml`. Listing 14-1 shows a typical declaration.

■ **Note** Remember, plug-in declarations come *last* in your `struts-config.xml` file. Placing it anywhere else will result in an exception being thrown when Struts starts up.

Listing 14-1. *Tiles Declaration in struts-config.xml*

```
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
   <set-property property="definitions-config"
                 value="/WEB-INF/tiles-defs.xml"/>
</plug-in>
```

The declaration in Listing 14-1 does two things. First, it tells Struts what *plug-in class* to instantiate. In this case, it's `org.apache.struts.tiles.TilesPlugin`. This plug-in class is responsible for the initialization of Struts.

Second, the `<plug-in>` tag contains one or more `<set-property>` tags, each of which passes a parameter-value pair to the plug-in class. In Listing 14-1, the only parameter passed to `org.apache.struts.tiles.TilesPlugin` is the `definitions-config` parameter, which points to the relative location of the Tiles definitions file. In this case, it's `/WEB-INF/tiles-defs.xml`.

This `tiles-def.xml` file (you may call it something else, if you wish—just change the `value` property if you do) is where you declare layouts and components for Tiles. You can think of it as the Tiles counterpart of `struts-config.xml`.

■ **Note** The Struts distribution contains a `tiles-documentation.war` file. It contains an example `tiles-def.xml` file you can use, in the `/WEB-INF/` folder of that WAR file.

Listing 14-2 shows the format of the `tiles-def.xml` file.

Listing 14-2. *A Blank Tiles Definitions File*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
      "http://struts.apache.org/dtds/tiles-config_1_1.dtd">

   <tiles-definitions>
<!-- one or more "definition" tags to
define layouts or components. -->
</tiles-definitions>
```

The root tag is `<tiles-definitions>`, and it contains one or more `<definition>` tags, with which you declare layouts or components. The next two sections show you how to do this.

■ **Note** The Tiles DTD (Document Tag Definition) is stored in the `struts.jar` file that comes with the Struts distribution. In this JAR file, the path to the DTD is `/org/apache/struts/resources/tiles-config_1_1.dtd`. This file also contains the `DOCTYPE` element used in Listing 14-2. It is also present in the `/lib` folder of the Struts distribution zip file.

You may split your Tiles definitions into more than one file. This is useful for large projects requiring several sets of look-and-feels or components that have to be maintained separately. Each such Tiles definitions file must follow the format of Listing 14-2, and you'll have to declare each file in your plug-in declaration. Simply separate the filenames with commas in the `<set-property>` tag. For example, if you had three Tiles definitions files, `A.xml`, `B.xml`, and `C.xml`, the `<set-property>` tag would read

```
<set-property property="definitions-config"
              value="/WEB-INF/A.xml,/WEB-INF/B.xml,/WEB-INF/C.xml"/>
```

Lastly, Tiles also uses a custom tag library, `struts-tiles.tld`. You will have to declare this in your `web.xml` file (refer to Chapter 3) in order to use this library on your JSPs.

Tiles for Layout

Layouts are a way to create a common look-and-feel for your web application. For example, you might want all JSP pages to have a footer containing copyright information, a side navigation bar, and a header with the company logo.

In Tiles, layouts are just JSP pages. The JSP page for a layout defines the relative placement of sections (body, header, footer, sidebars, etc.) on the displayed page. The content

of each section is defined by `<tiles:insert>` tags, which point to other HTML or JSP pages holding content. Listing 14-3 is a simple layout containing a title, header, body, and footer sections.

Listing 14-3. *simple-layout.jsp, a Simple Layout with Title, Header, Body, and Footer*

```
<%@ taglib uri="/tags/struts-tiles" prefix="tiles" %>
<html>
   <head>
      <title> <tiles:getAsString name="title"/> </title>
   </head>
   <body>
      <table>
           <tr><td> <tiles:insert attribute="header"/> </td></tr>
           <tr><td> <tiles:insert attribute="body"/> </td></tr>
           <tr><td> <tiles:insert attribute="footer"/> </td></tr>
      </table>
   </body>
</html>
```

As Listing 14-3 clearly demonstrates, a layout defines just the *relative placement* of content, not the actual content itself. Figure 14-1 shows the relative placement of the various components of the layout.

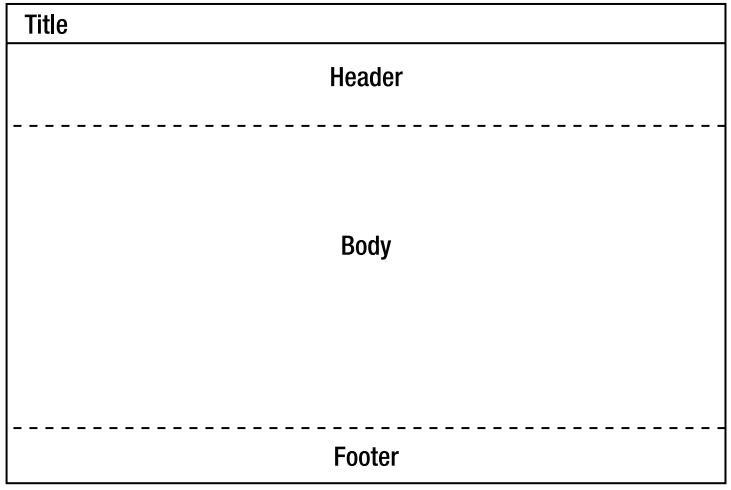

Figure 14-1. *The relative placement of header, body, and footer in simple-layout.jsp*

The <tiles:getAsString> tag displays content for the title, and the <tiles:insert> tags display the content corresponding to the attribute or name property. The actual content pointed to by these properties can be defined in either the called JSP or the Tiles definitions file.

For example, suppose myPage.jsp wanted to use the layout of Listing 14-3 (simple-layout.jsp). Listing 14-4 shows what the contents of myPage.jsp might be.

Listing 14-4. *myPage.jsp, Which Uses simple-layout.jsp*

```
<%@ taglib uri="/tags/struts-tiles" prefix="tiles" %>
<tiles:insert page="/layouts/simple-layout.jsp">
  <tiles:put name="title"  value="A simple page" />
  <tiles:put name="header" value="/common/header.jsp" />
  <tiles:put name="footer" value="/common/footer.jsp" />
  <tiles:put name="body"   value="/mypage-content.jsp" />
</tiles:insert>
```

All myPage.jsp does is specify (using the <tiles:put> nested tags) *which* files are to be associated with the header, footer, and body attributes. The title is specified as a string, in this case, "A simple page".

Notice the differences in how <tiles:insert> is used in Listings 14-3 and 14-4:

- In the layout JSP (Listing 14-3), the attribute property is used to tell Struts what content it should use to replace the enclosing <tiles:insert> tag. For example, when Struts processes <tiles:insert attribute="header"/> in Listing 14-3, it essentially looks for the value of header on the request object. It then uses this value (a JSP page, or as we will see in the next section, a component) and pastes the content from that page or component in place of the <tiles:insert> tag.

- In the called JSP (Listing 14-4), the page property is used to specify a layout, and the output from this layout is used to replace the <tiles:insert> tag. The nested <tiles:put> tags define Tiles attributes on the request object. The layout reads these attributes in order to create its output. For example, when Struts processes the <tiles:insert> in Listing 14-4, it knows it must use the layout file /layouts/simple-layout.jsp. In this layout file, there are a few <tiles:insert> tags that require the header, footer, and body attributes to be created. The values for each is defined in the calling JSP using <tiles:put> tags.

It is important to note that the attributes defined using a <tiles:put> tag will only apply to the layout defined in the enclosing <tiles:insert page="..."> tag. In the unlikely event that you had two <tiles:insert page="..."> tags on your called JSP page, you would have to specify the attributes for each separately. The attributes between these two tags are not shared.

■ **Note** The `page`, `template`, and `component` properties of the `<tiles:insert>` tag are synonymous. You might want to use a different synonym to indicate if a layout or Tiles component (see the next section) is used. This simply improves readability.

The other way to specify the values of the `attribute` or `name` property is to create a definition within the Tiles definitions file. For example, you could move part of Listing 14-4 into the Tiles definitions file, as shown in Listing 14-5.

Listing 14-5. *`tiles-def.xml`, with a Single Definition*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
      "http://struts.apache.org/dtds/tiles-config_1_1.dtd">

<tiles-definitions>
<definition name=".simple" path="/layouts/simple-layout.jsp">
<tiles:put name="header" value="/common/header.jsp" />
<tiles:put name="footer" value="/common/footer.jsp" />
</definition>
</tiles-definitions>
```

Listing 14-5 shows a Tiles definitions file called `.simple` (the meaning of the leading dot will be obvious shortly). This definition defines just two attributes, `header` and `footer`. To use the `.simple` definition in a JSP, see Listing 14-6.

Listing 14-6. *`myPage2.jsp`, Which Uses a Tiles Definition*

```
<%@ taglib uri="/tags/struts-tiles" prefix="tiles" %>
<tiles:insert definition=".simple">
  <tiles:put name="title"  value="A simple page" />
  <tiles:put name="body"   value="/mypage-content.jsp" />
</tiles:insert>
```

Listings 14-5 and 14-6 together show how to use a Tiles definition instead of a layout path. Shared attributes are the ones defined in the Tiles definitions file, while attributes specific to a called JSP page are defined on that page. This makes sense, because you might want to fix the header and footer but change the title and body of the displayed page.

Quick Quiz

What advantage does this approach (Listings 14-5 and 14-6) have over the earlier one (Listing 14-4)?

Tiles layout definitions can even be "subclassed." For example, suppose we wanted a new definition similar to the old one (in Listing 14-5), but where the title was fixed and the footer was changed. Listing 14-7 shows the amended definitions file.

Listing 14-7. *tiles-def.xml, with a New Definition*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
      "http://struts.apache.org/dtds/tiles-config_1_1.dtd">

  <tiles-definitions>
<definition name=".simple" path="/layouts/simple-layout.jsp">
<tiles:put name="header" value="/common/header.jsp" />
<tiles:put name="footer" value="/common/footer.jsp" />
</definition>
<definition name=".simple.admin" extends=".simple" >
<tiles:put name="title" value="Admin Zone" />
<tiles:put name="footer" value="/common/admin-footer.jsp" />
</definition>
</tiles-definitions>
```

The `extends` property indicates that the new definition named `.simple.admin` borrows all properties and attributes from the base `.simple` definition.

■ **Note** That's the rationale behind the `.` separator—you can tell at once that `.simple` has no ancestor definitions, and that `.simple.admin` descends directly from `.simple`.

The nested `<tiles:put>` tags either override certain attributes (`footer`) or add new ones (`title`). The new definition is used in exactly the same way as the old one.

The last way to define and use layouts is to define them *completely* in the Tiles definitions file. Continuing from Listing 14-7:

```
<definition name=".simple.admin.main" extends=".simple.admin" >
      <tiles:put name="body" value="/admin/body.jsp" />
</definition>
```

In this case, you have no need for a JSP page in which to embed the definition, since this new definition defines all attributes needed to display the layout. You may use this definition by calling it from either a `<forward>` in `struts-config.xml`:

```
<forward name="success" path=".simple.admin.main" />
```

or as an `input` page like so:

```
<action path="/MyFormHandler" input=".simple.admin.main" ...
```

Using Stylesheets with Layouts

Most professional web apps use Cascading Style Sheets (CSS) to achieve common fonts and colors. When you attempt to use stylesheets with Tiles layouts, you might run into problems. For example, suppose your stylesheet is `\styles\styles.css`, and your layout references this stylesheet, as shown in Listing 14-8.

Listing 14-8. *Snippet of `simple-layout.jsp` with Static Stylesheet Reference*

```
<%@ taglib uri="/tags/struts-tiles" prefix="tiles" %>
<html>
   <head>
      <style type="text/css" media="screen">
         @import "./styles/styles.css";
      </style>
      <title> <tiles:getAsString name="title"/> </title>
   </head>
...
```

This stylesheet declaration is fine if all your JSPs that use this layout are in the root folder of the web application. This may not be feasible for larger applications. For example, you might want to place "admin" pages in a separate `\admin\` subfolder. Doing so would mean that the reference to the stylesheet is now `../styles/styles.css` (note the leading `..`). The static stylesheet reference in Listing 14-8 won't do.

A useful hack is to use the `getContextPath()` function on the `request` object to get around this limitation. The stylesheet declaration of Listing 14-8 now becomes:

```
<style type="text/css" media="screen">
  @import "<%= request.getContextPath() %>/styles/styles.css";
</style>
```

Each time the layout is used, the right path to the stylesheet is computed. A poorer way of doing the same is to hard-code the full URL:

```
<style type="text/css" media="screen">
  @import "http://www.mycompany.com/myapp/styles/styles.css";
</style>
```

This method is less effective because the port used on the server or the application name might be changed upon deployment. In either case, the hard-coded solution won't work.

Tiles Components

In the previous sections, you've seen how to develop layouts using Tiles. Next, we'll explore a quite different use of Tiles: building components.

In Tiles, a *component* is a rectangular area displayed within the user's web browser (this is why "Tiles" is so named), capable of rendering *dynamic* content. Examples are components that display selected stock prices, or ads, or a user search list, or a list of pages the user has visited... The possibilities are endless!

There are two main points to note concerning Tiles components:

- **Easy embedding**: A Tiles component can be embedded into any JSP page using the handy `<tiles:insert>` tag. Want a weather report Tiles component on your JSP page? Slap in a single `<tiles:insert>` tag with the right attributes.

- **Independence**: To be useful, a Tiles component should function independently of its environment—the JSP page in which it is embedded. There are times when this rule can be broken (as you will see in the lab section of this chapter), but this is always at the expense of reusability.

Creating a Tiles Component

Creating a Tiles component is more involved compared to creating and declaring a Tiles layout. Five steps are involved:

1. **Create the Tiles *controller***. This is a Java class that creates dynamic content or processes user input.

2. **Declare the Tiles controller** in `struts-config.xml`.

3. **Create the Tiles "view."** This is the JSP page that displays the results of the Tiles controller's output.

4. **Declare a Tiles** `<definition>` that associates the JSP page with the controller. A controller may be used in more than one definition. This is exactly analogous to the way an `Action` subclass may be used in more than one form handler.

5. **Use the component** on your JSP pages by embedding the `<tiles:insert>` tag in them.

To create the Tiles controller, you subclass the `TilesAction` base class:

```
org.apache.struts.tiles.actions.TilesAction
```

Notice that it's `tiles.actions` (plural). I'll describe these five steps in more detail next and then wrap up with a concrete example application.

Step 1: Creating the Tiles Controller

The function you need to override is called (surprise!) `execute()`, but with a slightly different signature compared to `Action.execute()`, as shown in Listing 14-9.

Listing 14-9. *`execute()` on TilesAction*

```
public ActionForward execute(ComponentContext context,
                             ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
throws IOException, ServletException{ ...
```

All you have to do is to override this one function to process user input or prepare data for output.

The `org.apache.struts.tiles.ComponentContext` instance passed in `execute()` functions as a storage area for data intended for your Tiles component. You can store and retrieve data with `putAttribute()` and `getAttribute()` on this class. Only the JSP page represented by the `ActionForward` return value has direct access to data placed on this `ComponentContext` instance. This helps preserve the independent existence of the Tiles component from its environment.

To summarize, the primary purpose of the `ComponentContext` is as a channel of communication from the Tiles controller to the JSP page it forwards to.

This is analogous to the way the `HttpServletRequest` is a means of communication between an `Action` subclass and the forwarded JSP. A Tile controller could also place data on the `HttpServletRequest` object, but this would break the independence of that Tiles component, since other Tiles components on the page might overwrite the data placed on the `request` object.

The last point to note is that this `execute()` is marked to throw `IOException` and `javax.servlet.ServletException`.

Step 2: Declaring the Controller

The declaration for a Tiles controller is exactly the same as for any form handler. For example, Listing 14-10 shows a simple controller that doesn't accept any input data, and that has two possible views.

Listing 14-10. *One Controller, Two Views*

```
<action path="/MyTilesController"
        type="com.mycompany.myapp.tiles.MyTilesController">

   <forward   name="success"   path="/tiles/my-tiles-view.jsp"/>
   <forward   name="failure"   path="/tiles/error.jsp"/>
</action>
```

Perhaps the only difference is that when you declare a Tiles controller, it is *possible* to omit declaring any `<forward>`s, if there is only one JSP view associated with the Tiles component. Listing 14-11 shows an example.

Listing 14-11. *One Controller, One View*

```
<action path="/MyTilesController"
        type="com.mycompany.myapp.tiles.MyTilesController" />
```

In this case, the controller's `execute()` function (Listing 14-9) returns `null`, and the Tiles `<definition>` associated with this component is slightly different compared to the case where there are multiple JSP views associated with the Tiles component, as in Listing 14-10. I'll describe this difference shortly.

Step 3: Creating the Tiles View

The Tiles view is just the JSP that displays the output of the Tiles controller. This is analogous to the "next" JSP page associated with an `Action` subclass.

The primary difference, of course, is that the JSP page typically will evaluate to an HTML fragment, not a complete HTML page. This is because the JSP page's output will be embedded within a full HTML page.

Step 4: Declaring the Tiles <definition>

The Tiles `<definition>` enables the controller to be used *conveniently* in a `<tiles:insert>` tag. It is possible to use a Tiles controller already declared in `struts-config.xml` directly in a `<tiles:insert>`, just as a layout was used directly in Listing 14-4. For example:

```
<tiles:insert path="/MyTilesController.do"/>
```

The advantage of using a Tiles definition is that you can avoid hard-coding the handler's path in all your JSPs.

The declared definition will depend on whether the component uses just a single view page (as in Listing 14-11) or multiple views (Listing 14-10).

In the single view case, which is typical, the definition is declared as shown in Listing 14-12.

Listing 14-12. *A Tiles Component with a Single View*

```
<definition name=".myTile"
            path="/tiles/my-tiles-view.jsp"
            controllerUrl="/MyTilesController.do">

   <put name="title" value="This is My Tile"/>

</definition>
```

Declared this way, the return value of the controller's `execute()` should be `null`, and the JSP page in the `path` attribute is used as a view. The `controllerUrl` attribute is used to specify the Tiles handler declared in `struts-config.xml`. You may define attributes (in Listing 14-12, the `title` attribute is defined) to be used by the view.

If the component has multiple views, they must all be declared in `struts-config.xml` as `<forward>`s, as in Listing 14-10. In this case, the Tiles `<definition>` is simpler, as Listing 14-13 shows.

Listing 14-13. *A Tiles Component with Multiple Views*

```
<definition name=".myTile"
            path="/MyTilesController.do">

   <put name="title" value="This is My Tile"/>

</definition>
```

Notice that `path` now points to the Tiles controller declared in `struts-config.xml`. Again, you may define attributes to be used by any one of the views, using nested `<put>`s.

Step 5: Using the Component

As I said in the introduction to this section, using a tile is easy. All you have to do is to declare the use of the Tiles tag library in your JSP:

```
<%@ taglib uri="/tags/struts-tiles" prefix="tiles" %>
```

and then use a `<tiles:insert definition="...">` anywhere on your JSP page. For example, to use the Tiles component declared in either Listing 14-12 or Listing 14-13, use the following:

```
<tiles:insert definition=".myTile"/>
```

You can also specify further attributes for the view page by using nested `<tiles:put>` tags, as in Listing 14-6.

Example: The "Login" Tiles

Tiles components are very useful. Used wisely, they can greatly simplify your code and code maintenance.

Now, the discussion so far has been a little abstract, and rightly so, because there are a few ways to create and use Tiles components, and I wanted you to have a bird's-eye view of the whole thing.

In this section, I'll describe how our old friend the Registration webapp can be made into a Tiles component. Why do this? Because it gives you the freedom to place a user login anywhere on your webapp. If a user must log in several places in your webapp, then a Tiles component for user login would be a natural choice.

What I'll actually implement is a nontrivial variation on the Registration webapp. Figure 14-2 describes the control flow of this new souped-up version, perhaps better named a "Login" Tiles component.

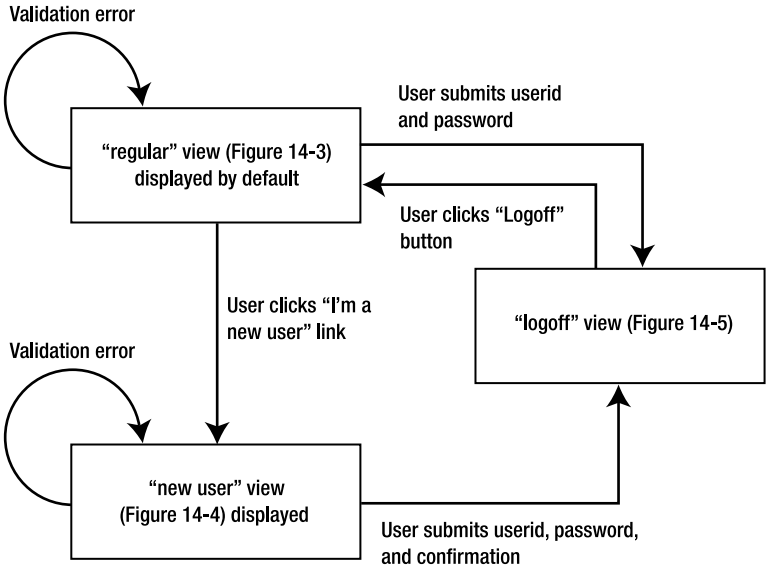

Figure 14-2. *Control flow of the Login component*

As you can see in Figure 14-2, there are three possible views:

- A "regular login" form with fields for user ID and password (see Figure 14-3)
- A "new user" form for users who don't have a user ID (Figure 14-4)
- A "logoff" view, displayed if a user has already logged on (Figure 14-5)

Also, unlike the Tiles components we've discussed thus far, this component accepts user input—for example, a user ID and password.

Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/login/
This is some text. This is more text. This is some text. This is some text. This is some text. This is some text. This is some text.
User ID
Password
Submit
I'm a new User
More blind text. Yet More blind text. More blind text. Yet More blind text. More blind text. Yet More blind text. More blind text. Yet More blind text.
http://localhost:8080/login/Login.do;jsessionid=EABB01ADFD6DFB49A4E21EB7FBDEC2CD?comman...

Figure 14-3. *The regular login form*

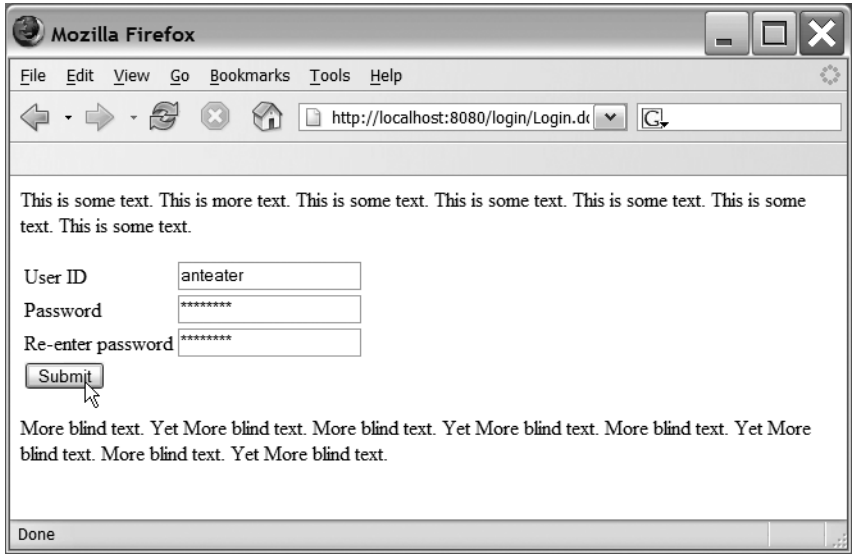
Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/login/Login.d
This is some text. This is more text. This is some text. This is some text. This is some text. This is some text. This is some text.
User ID
anteater
Password
****
Re-enter password
****
Submit
More blind text. Yet More blind text. More blind text. Yet More blind text. More blind text. Yet More blind text. More blind text. Yet More blind text.
Done

Figure 14-4. *The new user form*

Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/login/Login.d
This is some text. This is more text. This is some text. This is some text. This is some text. This is some text. This is some text.
You are logged on as anteater.
Logoff
More blind text. Yet More blind text. More blind text. Yet More blind text. More blind text. Yet More blind text. More blind text. Yet More blind text.
Done

Figure 14-5. *The logoff view*

Because this is a more complex component, for pedagogical reasons I'll describe each part of the component out of order, starting with the simplest.

First, Listing 14-14 shows how you'd embed the Login Tiles component in *any* JSP page.

Listing 14-14. *index.jsp, a JSP Page with the Login Component Embedded in It*

```
<%@ taglib uri="/tags/struts-tiles-el" prefix="tiles-el" %>
<html>
    <body>

    In this section, I'll describe how our old friend the ...

    <tiles-el:insert definition=".login"/>

    As you can see from Figure 14-2, there are ...

    </body>
</html>
```

As you can see from the taglib declaration, I'll be using the Struts EL-enabled tags (see Chapter 10).

Figures 14-3 and 14-4 both show that the Login component accepts user input. This data is stored in an `ActionForm` subclass called `LogonForm` (see Listing 14-15).

Listing 14-15. *LogonForm.java*

```
package net.thinksquared.login.struts;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public class LogonForm extends ActionForm {

  public static final int UNDEFINED = 0;
  public static final int LOGGED_ON = 1;
  public static final int LOG_OFF   = 2;
  public static final int REGULAR   = 3;
  public static final int NEW_USER  = 4;

  protected int    _status;
  protected String _userid;
  protected String _password;
  protected String _password2;

  protected boolean _hasErrors;
```

```
public LogonForm(){
  _status = UNDEFINED;
  _hasErrors = false;
}

public void   setStatus(int i){ _status = i; }
public int    getStatus(){ return _status; }

public void   setUserid(String s){ _userid = s; }
public String getUserid(){ return _userid; }

public void   setPassword(String s){ _password = s; }
public String getPassword(){ return _password; }

public void   setPassword2(String s){ _password2 = s; }
public String getPassword2(){ return _password2; }

public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request){

  if(_status >= REGULAR){

    ActionErrors errors = new ActionErrors();

    if(_userid == null || _userid.length() < 1){
      errors.add("userid",
         new ActionMessage("userlogin.error.userid.blank"));
    }else if(_password == null ||
             _password.length() < 6 || _password.length() > 12){
      errors.add("password",
         new ActionMessage("userlogin.error.password.length"));
    }else if(_status >= NEW_USER &&
            (_password2 == null || !_password2.equals(_password))){
      errors.add("password2",
         new ActionMessage("userlogin.error.password.retype"));
    }

    if(!errors.isEmpty()){
         _hasErrors = true;
         return errors;
    }
  }
```

```
    _hasErrors = false;
    return null;

}

  /***********  Status codes ************/
  public int getStatusRegular(){ return REGULAR;  }
  public int getStatusNewUser(){ return NEW_USER; }
  public int getStatusLogOff() { return LOG_OFF;  }

  /**********  Error status ************/
  public boolean hasErrors(){ return _hasErrors; }

  /**********  clear  form data ********/
  public void clear(){
    _status = UNDEFINED;
    _hasErrors = false;
    _userid = _password = _password2 = null;
  }

}
```

Listing 14-15 should hold no surprises, apart from the `hasErrors()` function and the three functions for status codes: `getStatusRegular()`, `getStatusNewUser()`, and `getStatusLogOff()`.The need for these functions will be apparent shortly.

Next, there are three views for the Login component. Listing 14-16 shows the "regular" view (Figure 14-3).

Listing 14-16. *regular.jsp*

```
<%@ taglib uri="/tags/struts-bean-el" prefix="bean-el" %>
<%@ taglib uri="/tags/struts-html-el" prefix="html-el" %>

<html-el:form action="/Login.do">

  <table>
  <tr>
    <td><bean-el:message key="userlogin.prompt.userid"/></td>
    <td>
        <html-el:text property="userid"/>
        <html-el:errors property="userid"/>
    </td>
  </tr>
  <tr>
```

```
   <td><bean-el:message key="userlogin.prompt.password"/></td>
   <td>
       <html-el:password property="password"/>
       <html-el:errors property="password"/>
   </td>
  </tr>
  <tr colspan="2">
   <td>
     <html-el:hidden property="status"
                     value="${LogonForm.statusRegular}"/>
     <html-el:submit>
       <bean-el:message key="userlogin.prompt.submit"/>
     </html-el:submit>
   </td>
  </tr>

  </table>
</html-el:form>

<html-el:link action="/Login.do?command=${LogonForm.statusNewUser}">
  <bean-el:message key="userlogin.prompt.new"/>
</html-el:link>
```

Listing 14-16 has a form and a link. The form accepts two fields: the user ID and password fields. The link allows new users to register with the system. This JSP page also explains how the "status codes" functions of `LogonForm.java` (Listing 14-15) are used. For example:

```
<html-el:hidden property="status"
                value="${LogonForm.statusRegular}"/>
```

uses EL to determine the value of the `regular` status code by calling the `getStatusRegular()` function on the `LogonForm`. The naive alternative would be to hard-code the value:

```
<html-el:hidden property="status" value="3"/> //don't do this!
```

Ugh! The "new user" view (Figure 14-4) is similar to the regular view, with the addition of the second "confirm password" field. Listing 14-17 shows the JSP code for this page.

Listing 14-17. *newuser.jsp*

```
<%@ taglib uri="/tags/struts-bean-el" prefix="bean-el" %>
<%@ taglib uri="/tags/struts-html-el" prefix="html-el" %>

<html-el:form action="/Login.do">
   <table>
     <tr>
        <td><bean-el:message key="userlogin.prompt.userid"/></td>
        <td>
            <html-el:text property="userid"/>
            <html-el:errors property="userid"/>
        </td>
     </tr>
     <tr>
        <td><bean-el:message key="userlogin.prompt.password"/></td>
        <td>
            <html-el:password property="password"/>
            <html-el:errors property="password"/>
        </td>
     </tr>
     <tr>
        <td><bean-el:message key="userlogin.prompt.password2"/></td>
        <td>
            <html-el:password property="password2"/>
            <html-el:errors property="password2"/>
        </td>
     </tr>
     <tr colspan="2">
        <td>
           <html-el:hidden property="status"
              value="${LogonForm.statusNewUser}"/>

           <html-el:submit>
                <bean-el:message key="userlogin.prompt.submit"/>
           </html-el:submit>
        </td>
     </tr>
   </table>
</html-el:form>
```

Lastly, the "logoff" view (see Listing 14-18) has just a single logoff button, as Figure 14-5 shows.

Listing 14-18. *success.jsp*

```
<%@ taglib uri="/tags/struts-bean-el" prefix="bean-el" %>
<%@ taglib uri="/tags/struts-html-el" prefix="html-el" %>

<html-el:form action="/Logoff.do">
    <bean-el:message key="userlogin.prompt.loggedonas"
                    arg0="${LogonForm.userid}"/>

   <html-el:submit>
       <bean-el:message key="userlogin.prompt.logoff"/>
   </html-el:submit>

   <html-el:hidden property="status"
                  value="${LogonForm.statusLogOff}"/>

</html-el:form>
```

Notice that the logoff view (`success.jsp`) submits data to the form handler called `Logoff.do`, unlike the previous two views, which submitted data to `Login.do`. This is because we don't need to perform validation for a logoff, so the declaration for `Logoff.do` in `struts-config.xml` has `validate="false"` and no input page.

Next, we examine the `struts-config.xml` file (see Listing 14-19). For clarity, I'll only show the `form-beans` and `action-mappings` sections.

Listing 14-19. *form-beans and action-mappings Sections of `struts-config.xml`*

```
<form-beans>
  <form-bean name="LogonForm"
             type="net.thinksquared.login.struts.LogonForm"/>
</form-beans>

<action-mappings>

  <action path="/Login"
          type="org.apache.struts.actions.ForwardAction"
          name="LogonForm"
          scope="session"
          validate="true"
          input="/index.jsp"
          parameter="/index.jsp"/>
```

```
  <action path="/Logoff"
          type="org.apache.struts.actions.ForwardAction"
          name="LogonForm"
          scope="session"
          validate="false"
          parameter="/index.jsp"/>

  <action path="/LoginController"
          type="net.thinksquared.login.struts.UserLoginAction"
          name="LogonForm"
          scope="session"
          validate="false">

     <forward name="success"  path="/userlogin/success.jsp"/>
     <forward name="new-user" path="/userlogin/newuser.jsp"/>
     <forward name="regular"  path="/userlogin/regular.jsp"/>
   </action>

</action-mappings>
```

The declarations here are a little tricky, so be sure to pay careful attention. First, the `LogonForm` is declared. This should be familiar to you and doesn't require further explanation.

Next come the three form handlers. The first two, `Login.do` and `Logoff.do`, are called directly by the forms on the three views. The regular and new user views submit to `Login.do`, and the logoff view submits to `Logoff.do`.

The only difference between the two form handlers is that `Logoff.do` does not validate the form data (since we're logging off).

Both `Login.do` and `Logoff.do` are associated with `ForwardAction`, which forwards to the page specified in the `parameter` attribute of the form handler. This `parameter` attribute is *required* for `ForwardAction`. No processing is done by `ForwardAction` apart from this.

So, from the declarations in Listing 14-19, it is apparent that both `Login.do` and `Logoff.do` do nothing but immediately redisplay the input page!

This might seem a little strange. Why immediately redisplay a submitted page with apparently no processing? The answer, of course, is that the input page (Listing 14-14) contains an embedded `Login <tiles:insert>`. When the input page is redisplayed, the embedded Tiles component is updated too. In this updating, the associated Tiles controller is called, and the form data is actually processed.

We see from Listing 14-14 that the called definition is `.login`. The declaration for this definition is

```
<definition name=".login" path="/LoginController.do" />
```

So, the Tiles controller is the form handler called `LoginController.do`:

```
<action path="/LoginController"
        type="net.thinksquared.login.struts.UserLoginAction"
        name="LogonForm"
        scope="session"
        validate="false">

  <forward name="success"  path="/userlogin/success.jsp"/>
  <forward name="new-user" path="/userlogin/newuser.jsp"/>
  <forward name="regular"  path="/userlogin/regular.jsp"/>
</action>
```

`UserLoginAction` is a `TilesAction` subclass, which I'll describe shortly. `LoginController` itself performs *no* simple validation (`validate="false"`), which should not raise any eyebrows, since simple validation has been done by the main `Login.do` form handler for the page.

All the form handlers—`Login.do`, `Logoff.do`, and `LoginController.do`—are all set for session scope. This makes sense because we want the login information to persist for as long as the user interacts with the system. The added bonus to this is that if *another* JSP page embeds the Login component, the correct view will be displayed. The user need only log on once, and all other pages with the Login component will correctly reflect this too.

Lastly, we come to the `UserLoginAction` class (see Listing 14-20), where the processing of user data and flow control is done.

Listing 14-20. *UserLoginAction.java*

```
package net.thinksquared.login.struts;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.struts.tiles.*;
import org.apache.struts.tiles.actions.*;
import org.apache.struts.action.*;

public final class UserLoginAction extends TilesAction{

  public ActionForward execute(ComponentContext ctx,
                               ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response)
  throws IOException,ServletException{
```

```
LogonForm lForm = (LogonForm) form;

String cmd = request.getParameter("command");
if(cmd != null){
  return mapping.findForward("new-user");
}

if(lForm.getStatus() == LogonForm.UNDEFINED){
  return mapping.findForward("regular");
}

if(lForm.getStatus() == LogonForm.LOGGED_ON){
  return mapping.findForward("success");
}

if(lForm.getStatus() == LogonForm.LOG_OFF){
  System.out.println("Logging off " + lForm.getUserid());
  lForm.clear();
  return mapping.findForward("regular");
}

if(lForm.getStatus() == LogonForm.NEW_USER){
  if(lForm.hasErrors()){
     return mapping.findForward("new-user");
  }else{
     System.out.println("Creating New User " + lForm.getUserid());
     return mapping.findForward("success");
  }
}

if(lForm.getStatus() == LogonForm.REGULAR){
  if(lForm.hasErrors()){
    return mapping.findForward("regular");
  }else{
    System.out.println("Logging on regular user " +
                       lForm.getUserid());
    return mapping.findForward("success");
  }
}
```

```
    //catch-all
    return mapping.findForward("regular");
  }
}
```

The first thing we do is check if there are any parameters on the URL:

```
String cmd = request.getParameter("command");
if(cmd != null){
  return mapping.findForward("new-user");
}
```

The only reason why there might be a parameter on the URL is if the user clicked the "I'm a new user" link on `regular.jsp` (see Listing 14-21).

Listing 14-21. *Link Snippet from `regular.jsp`*

```
<html-el:link action="/Login.do?command=${LogonForm.statusNewUser}">
  <bean-el:message key="userlogin.prompt.new"/>
</html-el:link>
```

In this case, we obviously want to display the new user view.

■ **Note** Since we're merely checking for the presence of a `command` parameter, we could have just used `<html-el:link action="/Login.do?command="dummy"..>`. However, if we add more linked views in the future, we'll have to change the link to something along the lines of Listing 14-21.

The next two blocks of code simply redisplay the necessary views:

```
if(lForm.getStatus() == LogonForm.UNDEFINED){
  return mapping.findForward("regular");
}

if(lForm.getStatus() == LogonForm.LOGGED_ON){
  return mapping.findForward("success");
}
```

This is needed when the page first loads (the first block) or if the user refreshes the page after having logged on (second block).

The following block logs off the user, taking care to clear the form before the default regular view is displayed:

```
if(lForm.getStatus() == LogonForm.LOG_OFF){
  System.out.println("Logging off " + lForm.getUserid());
  lForm.clear();
  return mapping.findForward("regular");
}
```

For clarity, I haven't implemented the model code but used `System.out.println()` calls to indicate what should be done.

The last two blocks are perhaps the most interesting:

```
if(lForm.getStatus() == LogonForm.NEW_USER){
  if(lForm.hasErrors()){
    return mapping.findForward("new-user");
  }else{
    System.out.println("Creating New User " + lForm.getUserid());
    return mapping.findForward("success");
  }
}

if(lForm.getStatus() == LogonForm.REGULAR){
 if(lForm.hasErrors()){
   return mapping.findForward("regular");
 }else{
   System.out.println("Logging on regular user " + lForm.getUserid());
   return mapping.findForward("success");
 }
}
```

All these blocks of code do is display the logoff view, after the user correctly fills in either the regular login form or the new user login form. But notice that the form is checked for errors with `hasErrors()` (see Listing 14-15) before the logoff view is displayed.

This is the reason we created the `hasErrors()` function on the `LogonForm`—it enables the Login component to tell whether the form data passed simple validation.

We can't rely on Struts automatically redisplaying a page upon simple validation failure, because the JSP for the Login Tiles component doesn't exist when the page is redisplayed. It's the Tiles controller (`UserLoginAction`) that decides which JSP to paste into the input page. So it has to know if there are validation errors in order to paste the right JSP. Creating a `hasErrors()` function on the `LogonForm` is a simple way of doing this.

That pretty much wraps up the Login Tiles component example. Please visit the Source Code section of the Apress website at `http://www.apress.com` to download the source code for the Login Tiles component. The WAR file, ready for deployment, is called `login.war`, and it's also in the Source Code section of the Apress website. I encourage you to experiment with this simple Tiles component to gain a better understanding of how Tiles components work.

Getting External Form Data

What if there was more than one Tiles component on the page? Would the cast

```
LogonForm lForm = (LogonForm) form;
```

in Listing 14-20 throw an exception? The answer is no, because the type of `ActionForm` (more precisely, the form bean) passed to the Tiles controller must match the one declared in `struts-config.xml`. If no such form bean was submitted, then a blank one is created, and passed to the Tiles controller. This means the cast will *always work* as long as the declaration is correct.

This has an interesting implication: suppose form data is sent to a form handler associated with form bean X. Then, *all* Tiles components on that same page that also are associated with that form bean X will receive carbon copies of that form data.

For example, consider the following set of form-handler declarations:

```
<action path="MyAction"           name="MyFormBean" type="...">
<action path="MyTileControllerA"  name="MyFormBean" type="...">
<action path="MyTileControllerB"  name="MyFormBean" type="...">
<action path="MyTileControllerC"  name="DifferentFormBean" type="...">
```

and the following JSP page snippet:

```
<html:form action="/MyAction.do">
   ...
</html:form>
<tiles:insert definition=".myTilesComponentA"/>
<tiles:insert definition=".myTilesComponentB"/>
<tiles:insert definition=".myTilesComponentC"/>
```

When the form data is submitted, it is first stored in an instance of `MyFormBean` and then processed by the `Action` subclass associated with `MyAction`.

What is less obvious is that the `execute()` functions on `MyTileControllerA` and `MyTileControllerB` are passed *separate copies* of `MyFormBean`. The `execute()` function of `MyTilesControllerC` is passed a new instance of `DifferentFormBean`.

You will use this trick in the following lab session, to get a Tiles component to receive data from the external form.

Lab 14: The Find Facility

We want to add a Find function to both `full.jsp` and `mnc.jsp`, making minimal alterations to existing code. Specifically:

- We want the existing form in both JSPs to be used to deliver keyword search into our Find utility.
- If the user clicks Find, the form data is used as keywords. A list of contacts that matches the query appears. Each contact is an HTML link.
- If the user clicks on a contact link, the form populates with the contact's data. If the user clicks Submit, the form data is interpreted as an update to the contact information.
- If the user just fills in the form and clicks Submit instead of Find, the form data is interpreted as a new entry.

Figure 14-6 shows the placement of the Find facility within the `full.jsp` and `mnc.jsp` pages.

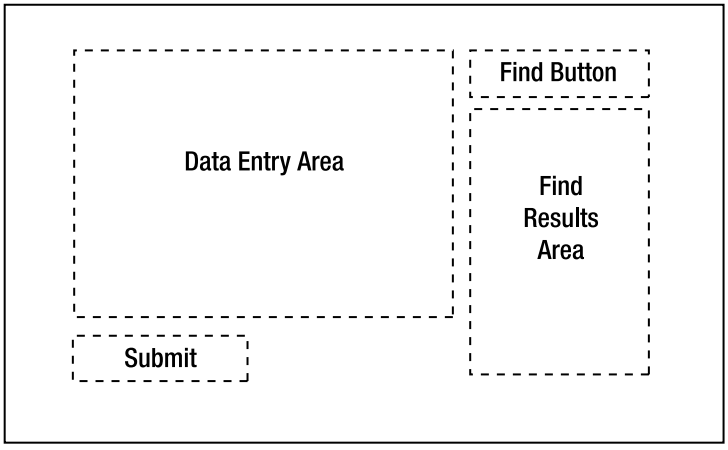

Figure 14-6. *Placement of the Find facility within the existing JSP pages*

Notice that the form now has two submit buttons: one to submit form data (the "true" submit button) and another to submit the keyed-in data as a search query (the Find button).

The best way to tackle this is to use the `LookupDispatchAction` technique (see Chapter 17). Please read that the section on `LookupDispatchAction` before proceeding.

This lab also makes use of Lisptorq-generated Model classes, so be sure to read the relevant sections in Appendix A first.

Step 1: Set Up Tiles

1. Put in the plug-in section for Tiles on `struts-config.xml`. Remember to put in the plug-in section at the end of the file. Call the Tiles definitions file `tiles-def.xml`.

2. Tiles uses a tag library called `struts-tiles.tld`. Declare this in `web.xml`.

Step 2: Write the Controller

The class `ContactPeer` has a `find()` function:

```
public static Scroller find(Contact query)
```

You will need this function to perform searches based on a given `Contact`.

Now, the relevant `Controller` class for the Find tile is `FindAction`, which is a subclass of `net.thinksquarded.lilldep.struts.TilesLookupDispatchAction`. This base class is used much in the same way as `LookupDispatchAction` (see Chapter 17), but is specifically for Tiles.

1. Complete the implementation of `find()` on `FindAction`, so that it extracts the `Contact` instance from the given `ActionForm`. Save a *cloned* version of this `Contact` in the current session, under the key `JSPConstants.FIND_QUERY`. Use `Contact.klone()`, which clones the `Contact`.

2. Use `ContactPeer.find()` to create a `Scroller` object using this `Contact`. You should place this `Scroller` object in the current request, under the key `JspConstants.FIND_RESULTS`.

3. `find()` should return `null`, because navigation is determined by the main handler for the form.

4. Complete the implementation of `unknown()` on `FindAction`, so that it checks the session for a `Contact`, under the key `JspConstants.FIND_QUERY`. If the `Contact` exists, then use it to get a `Scroller` using `ContactPeer.find()`. Save the `Scroller` on the current request. `unknown()` should return `null`.

5. Implement the function `getKeyMethodMap()`, which has the same requirements as `LookupDispatchAction.getKeyMethodMap()`.

6. Use `compile.bat` to see if your work compiles.

I hope you see what we're doing here. We want to save the query in session scope so that we can view the Find results from page to page. We can't simply save the `Scroller` (and use `absolute()` to reset it each time we need it) because we might change the `Contact`'s details so that it no longer matches the query.

There are a few subtleties in this step:

- Notice that we have to save a clone of the query, and not the query itself. This is because Struts resets the fields on the query object itself. This behavior is part of how Struts works. So, to avoid holding a blanked-out form, we have to clone it.

- The `unknown()` function is called (you can see the behavior from `TilesLookupDispatchAction`) when the `command` parameter in the URL points to a function that does not exist on the Tiles `Controller` (i.e., `FindAction`). In our case, `unknown()` will be called when the user clicks Submit instead of Find.

- When the `full.jsp` page is called directly or by other form handlers (e.g., `EditContact.do`), then the `command` parameter isn't present in the URL. This means that the `unspecified()` function on `TilesLookupDispatchAction` is called. So, in order for the Find utility to correctly update itself, `unspecified()` has to call `unknown()`.

Step 3: Put In the Tiles Action Mapping

1. Create a new form handler in `struts-config.xml` to declare the Tiles controller you just created.

2. Set the `parameter` attribute to `command`. This is necessary for `TilesLookupDispatchAction` to correctly dispatch according to the submit button that was clicked.

3. Set the scope of the form handler to `session`, in order to match that of the main form.

4. Be sure that your Tiles controller will get a copy of the form submitted from the main page. (See the section "Getting External Form Data" to see how this is done.)

Step 4: Make Changes to ContactAction

Since there are now *two* submit buttons (Submit and Find), we require `ContactAction` to process Submit but ignore Find.

1. We'll use `LookupDispatchAction` to do this, so you'll have to change the base class of `ContactAction` to `LookupDispatchAction`.

2. Change the name of `execute()` to `save()`.

3. Because there are now two submit buttons, we can't rely on Struts to automatically validate the form for us. This means that we have to call the validate function from `save()` instead. Make the necessary changes to do this, taking care to set `validate=false` on the associated form handlers declared in `struts-config.xml`.

4. Implement the function `find()`, which has the same signature and return type as `execute()`. This function should replace the form's `Contact` with a clean instance and then return the forward called `success`.

5. Implement `getKeyMethodMap()`.

You also need to add an extra attribute called `parameter`, whose value is set to `command`, to the form handlers associated with `ContactAction` (there are two of them). This required for `LookupDispatchAction` to correctly dispatch according to the submit button clicked.

Step 5: Write the Tiles JSP

1. Edit `find-tile.jsp` so that it displays the company name using the `Scroller` saved on the request by the name `JspConstants.FIND`. Hint: Use `<logic:iterate>` and simply iterate through, as you did for `listing.jsp`. Remember to check for a `null Scroller`.

2. Make a link to `EditAction` for each company name, as you did in `listing.jsp` in Chapter 13.

3. You get extra credit if you can get the background of the company name link to change color when it's clicked (no JavaScript solutions—use Struts).

Step 6: Write the Tiles Definition

Add a new definition in the Tiles definitions file, `tiles-def.xml`. The definition should do the following:

- Provide a unique name to the definition.

- Link the Tiles form handler in step 2 to the Tiles JSP in step 4.

Step 7: Put In the Find Tile

Now, you need to put the Find tile in your JSPs. We'll start with `full.jsp`:

1. Look at the layout specs (Figure 14-6), and put in a table with two cells. One cell will hold the existing data entry panel, and the other (on the right) will hold the Find button and the results window.

2. Put in a Find button (look up the key in `Application.properties`!), which is an `<html:submit>`.

3. You need to add `property="command"` to the new Find button as well as the existing Submit button.

4. Put in the taglib declaration for the Tiles tag library.

5. Repeat steps 1–3 for `mnc.jsp`.

Step 8: Deploy and Test

Compile and deploy your webapp. Verify the following:

- Does the Find utility work? In other words, do you see a list of companies when you use Find?
- Does the form populate when you click the company name links?
- Does the company name's background color change when you click on it?
- Does the contact update correctly when you make a change?
- Does the Find result change when you make an update? For example, if you perform a Find operation for a company name, and then change the company name, does the Find listing change?
- Does Find work correctly on both `full.jsp` and `mnc.jsp`?
- If you use the full/MNC entry forms to enter a new contact, does the database correctly create new contacts?

Summary

- Tiles is a mechanism for creating layouts and components.
- Layouts help streamline a web application's look and feel.
- Tiles components are a way to create reusable GUI components.

CHAPTER 15

The Validator Framework

The Validator framework is a Struts plug-in, originally created by David Winterfeldt, that provides a set of *generic validations* that you can use to run a variety of simple validations. Instead of implementing `validate()` functions in your `ActionForm` subclasses, you declare your validations in an XML file.

So, the Validator framework eliminates the need for implementing `validate()` in your `ActionForm` subclasses. (You still need your `ActionForm` subclasses to store form data.)

You reap a number of benefits when you use the Validator framework, as opposed to writing individual `validate()` functions on your `ActionForm` subclasses:

- **Better maintainability**: Since the new XML format validations are all placed in one file, you should be able to maintain your validations more easily.

- **Standardization**: Many simple validations require the same checks. For example, both `userid` and `password` fields might require checks to ensure they are composed of alphanumerics. If these checks were spread throughout a few `validate()` functions, standardizing them would be a challenge. As you'll see, the Validator framework makes it easier for you to standardize your validations.

- **Correct implementation**: Some "simple" validations are quite difficult to implement correctly! A prime example is validating an email address so that it conforms to the RFC-2822 specification (this specification deals with email address formats, among other things). Rather than reinventing the wheel, you can take advantage of validations provided by the Validator framework.

- **Less code duplication**: Similarly, because the Validator framework provides a fairly comprehensive set of validations, you can cut down code duplication. You don't need to implement your own code to validate things like credit card numbers in each of your web applications. You could argue that it's possible to write these validations once, put them in a single class, and reuse them for all your apps. True, but this means that new members of your team would have more to learn before they could become productive. In the worst-case scenario, some developers might forget about the shared library of validations and duplicate validations in their `validate()` functions.

- **Automatic client-side validations**: You can automatically generate JavaScript validations on your client by placing a single `<html:javascript/>` tag in your JSPs. The validator framework will automatically generate the validations for you. This feature of the Validation framework is still underdeveloped; there's no way for you to specify client-side validation *only*.

The set of generic validations provided by the Validator framework is quite comprehensive, and you should be able to replace all your `validate()` functions with XML declarations. However, in case you can't, it's possible to run your own custom validations *alongside* the ones you've declared using the Validator framework. It's also possible to *extend* the set of standard validations provided by the Validator framework.

Declaring the Validator Plug-in

Like Tiles (see Chapter 14), the Validator framework is independent of Struts. It belongs to the Apache Commons project (see "Useful Links" at the end of this chapter), which aims to create a set of reusable frameworks, the Validator framework being one of them.

Indeed, the Validator framework internals are stored in the `commons-validator.jar` file that comes with the Struts distribution. This JAR file contains all the Java classes for the framework and the JavaScript to run client-side validations.

To use the Validator framework in your Struts applications, you need to declare it. This means a `<plug-in>` declaration in the `struts-config.xml` file (see Listing 15-1).

Listing 15-1. *Plug-in Declaration for the Validator Framework in struts-config.xml*

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn" >
  <set-property property="pathnames"
       value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

Notice that it's `ValidatorPlugIn` (capital I). A common source of frustration is to use the lowercase "i" as it is in `TilesPlugin`.

The `<set-property>` tag sets a single property called `pathnames` accessible to the plug-in. The value of `pathnames` is a set of comma-separated files:

- `validator-rules.xml` is a set of declared generic validations for the Validator framework. This file is provided for you in the Struts distribution.

- `validation.xml` links each field in your `ActionForm` subclass (more precisely, the declared form bean) to one or more validations drawn from those declared in `validator-rules.xml`. So, `validation.xml` is where you replace the `validate()` functions of your `ActionForm` subclasses.

As the declaration suggests, you need to place both these files in the /WEB-INF/ folder of your web application.

Both validator-rules.xml and validation.xml have the same structure—that is, they are described by the same XML document tag definition (DTD).

■ **Note** The DTD is contained in the commons-validator.jar file, which comes with the Struts distribution, under the path /org/apache/commons/resources/validator_1_1_3.dtd.

Since they have the same structure, there's no reason why you can't create a single file containing information from both. Of course, this leads to maintenance issues.

Alternatively, you could split your validation.xml file into several files. To use them, just change the plug-in declaration. For example, if you had three validation files, called A.xml, B.xml, and C.xml, then the code Listing 15-1 would be changed to that shown in Listing 15-2. Struts pieces together all *four* files declared in Listing 15-2 before processing them.

Listing 15-2. *Splitting Your validation.xml file into Three Parts*

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn" >
  <set-property property="pathnames"
       value="/WEB-INF/validator-rules.xml,
              /WEB-INF/A.xml,
              /WEB-INF/B.xml,
              /WEB-INF/C.xml"/>
</plug-in>
```

Splitting your validation.xml file is useful if your project contains subprojects. It is easier to manage and maintain a few smaller files, each containing forms from a single subproject, than one huge file containing validations for all projects.

Validator DTD Basics

Since the validator-rules.xml and validation.xml files play a central role in the Validator framework, you have to understand their structure. Listing 15-3 is a fragment of the validator DTD, and it describes the major divisions of the validator XML structure.

Listing 15-3. *Top Nodes of the Validator XML Structure*

```
<!ELEMENT form-validation (global*, formset*)>
<!ELEMENT global (validator*, constant*)>
<!ELEMENT formset (constant*, form+)>
```

The <form-validation> tag is the root tag, and all other tags are contained within it. The important subtags of <form-validation> are

- <global>, which in turn contains declarations of "global" constants (we'll come to this shortly) or declarations for validators.
- <formset>, which represents validations for a locale. We'll discuss this at the end of this chapter.
- <form>, which is a subtag of <formset>, which contains your validations of fields on any form beans declared in struts-config.xml. You will declare these in your validation.xml file.

In the validator-rules.xml file, you will only see <validator> declarations, one for each generic validator available on the Validator framework.

Your validation.xml file should contain only <form> or <constant> declarations. This division of tags between validator-rules.xml and validation.xml is purely for maintainability: you put all declared validations in validator-rules.xml, and your *use* of the validations in validation.xml.

I'll describe these tags in more detail as I show you how to use and extend the Validator framework.

Using the Validator Framework

Using the Validator framework is easy:

- For each form bean needing validation, you add a new <form> tag to your validation.xml file. The name attribute of the <form> should be the same as the name of the form bean.
- For each field requiring validation in the form bean, you add a <field> subtag to the <form>. The property attribute of the <field> tag should be the same as the name of the field you want to validate. The property attribute supports nested fields (see Chapter 10 for details on nested fields/properties).
- The <field> tag also has a depends attribute, with which you specify the validations you want to run for this field. The names of the validations are specified as a comma-delimited list.
- The <field> tag may contain optional subtags for custom error messages or variables used by a validator (e.g., minimum length of a field).
- Ensure that your form bean is a subclass of org.apache.struts.validator.ValidatorForm.

That's it! To see how this works, let's explore how you'd use the Validator framework to validate `RegistrationForm`, the only form bean for the Registration webapp (see Listing 6-1 in Chapter 6).

Example: Validating RegistrationForm

`RegistrationForm` (refer to Listing 6-1) has three fields that need to be validated. In this section, I'll create a version of `RegistrationForm` that uses the Validator framework. The new `RegistrationForm.java` class is shown in Listing 15-4.

Listing 15-4. *RegistrationForm.java for the Validator Framework*

```
package net.thinksquared.registration.struts;

import org.apache.struts.validator.*;

public final class RegistrationForm extends ValidatorForm{

   private String _userid = null;
   private String _pwd = null;
   private String _pwd2 = null;

   /**
    *   getXXX and setXXX functions
    *   corresponding to form properties
   */
   public String getUserid(){ return _userid; }
   public void setUserid(String userid){ _userid = userid; }

   public String getPassword(){ return _pwd; }
   public void setPassword(String pwd){ _pwd = pwd; }

   public String getPassword2(){ return _pwd2; }
   public void setPassword2(String pwd2){ _pwd2 = pwd2; }

}
```

If you compare Listing 15-4 with Listing 6-1, you'll notice a couple of changes:

- `RegistrationForm` now extends `org.apache.struts.validator.ValidatorForm` instead of `org.apache.struts.action.ActionForm`.
- The `validate()` function has disappeared.

You need to subclass `ValidatorForm` because this base class will run validations declared in your `validation.xml` file. This also explains the disappearance of the `validate()` function—Struts will call `validate()` on `ValidatorForm` instead.

■ **Note** The form bean declaration for `RegistrationForm` would be unchanged.

To understand how to declare validations, consider that `RegistrationForm` has three fields: `userid`, `password`, and `password2`. The checks run in Listing 6-1 are to ensure that the `userid` is filled in and that `password` and `password2` match.

These validations are hardly complete. We'll add a max/min check for the `userid` and `password`, and a check that `userid` and `password` are restricted to alphanumerics. The `validation.xml` file incorporating all these checks appears in Listing 15-5.

Listing 15-5. *Declaring Validations for RegistrationForm*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE form-validation PUBLIC ... // truncated for clarity
<form-validation>
   <formset>

      <form name="RegistrationForm">
          <field property="userid"
                  depends="required,mask,maxLength,minLength">

             <msg name="required"  key="reg.error.userid.missing" />
             <msg name="mask"      key="reg.error.userid.alphanum"/>
             <msg name="minLength" key="reg.error.userid.length"  />
             <msg name="maxLength" key="reg.error.userid.length"  />

             <arg name="minLength" key="${var:minlength}" position="0"
                  resource="false"/>
             <arg name="minLength" key="${var:maxlength}" position="1"
                  resource="false"/>

             <arg name="maxLength" key="${var:minlength}" position="0"
                  resource="false"/>
             <arg name="maxLength" key="${var:maxlength}" position="1"
                  resource="false"/>
```

```
            <var>
               <var-name>mask</var-name>
               <var-value>^[a-zA-Z0-9]*$</var-value>
            </var>
            <var>
               <var-name>minlength</var-name>
               <var-value>5</var-value>
            </var>
            <var>
               <var-name>maxlength</var-name>
               <var-value>10</var-value>
            </var>
         </field>

         <field property="password"
                depends="required,mask,maxLength,minLength">

           // similar to validations for userid,
           // omitted for clarity.

         </field>

         <field property="password2" depends="validwhen">
            <msg name="validwhen" key="reg.error.password.mismatch"/>
            <var>
               <var-name>test</var-name>
               <var-value>(password == *this*)</var-value>
            </var>
         </field>

      </form>
   </formset>
</form-validation>
```

As you can see, there's just one `<formset>` tag and this means that we haven't localized any validations. Don't get me wrong—the *error messages* are localized, since Listing 15-5 obviously uses keys from the properties file. But the validations themselves are not localized. To understand the difference, refer to the discussion on "Localizing Validations" in Chapter 12.

Most of the time, you won't be localizing validations, so your `validation.xml` file will contain just one `<formset>` tag. I'll describe how to use multiple `<formset>` tags to help you localize validations toward the end of this chapter.

The nested <form> tag is where the action happens. In Listing 15-5, there's just one <form> tag, corresponding to the RegistrationForm form bean that needs validating. If you had other form beans that needed validating, then the <formset> would contain more <form> tags, one for each of these form beans.

The validations for each field are done separately. The validations for the password field are very similar to the ones for userid, so I've left this field out. From the <field> declaration for the userid field, you can tell I've run four validations:

```
<field property="userid"
       depends="required,mask,maxLength,minLength">
```

The required validator checks that the field is filled in by the user. It requires no extra information to run. If the required validator fails, the following <msg> tag

```
<msg name="required"  key="reg.error.userid.missing"/>
```

ensures that the error message specified by the key reg.error.userid.missing is displayed. It's possible (but not advisable) to force the display of a static string using the resource=false attribute:

```
<msg name="required"  key="Userid is missing!" resource="false"/>
```

The mask validator checks that the field's value matches a given regular expression.

■ **Note** A regular expression is a way to compactly express any string. For example, the DOS regular expression *.* matches all filenames with an extension. Regular expression languages vary by provider, so *.* will *not* match every filename with an extension if used in the Validator framework. The Validator framework uses a much more powerful regular expression language, similar to the one used in the Perl programming language. Refer to the "Useful Links" section at the end of this chapter for a reference.

This regular expression is given in a var tag:

```
<var>
   <var-name>mask</var-name>
   <var-value>^[a-zA-Z0-9]*$</var-value>
</var>
```

This declares a *variable* named mask, with the value ^[a-zA-Z0-9]*$. The mask *validator* requires this variable to be defined in order to work. In this case, the regular expression matches alphanumeric strings (strings containing roman alphabets and Arabic numerals, 0 through 9).

Similarly, the maxLength and minLength validators require variables maxlength and minlength, respectively.

The error messages for these checks require some explanation. For example, the custom error message for `minLength` is declared by

```
<msg name="minLength" key="reg.error.userid.length"/>
```

The entry in `Application.properties` for this key is

```
Passwords should be between {0} and {1} characters long
```

The `{0}` and `{1}` replacement arguments have to be defined using `<arg>` tags:

```
<arg name="minLength" key="${var:minlength}" position="0"
     resource="false"/>
<arg name="minLength" key="${var:maxlength}" position="1"
     resource="false"/>
```

The `name` attribute corresponds to the name of the validator (this is also present in the `<msg>` tag), and the `key` in this case refers to the `minlength` and `maxlength` variable values. Since the key attribute is to be interpreted as a static value and not as a key in `Application.properties`, I've set `resource="false"`. The `position` attribute indicates which replacement argument this refers to.

LOCALIZING THE REPLACEMENT ARGUMENTS

In Listing 15-5, the replacement arguments are *not* localized. So, a user in the Arabic locale would see the error message in Arabic *except* for the numerals (the modern Arabic numerals are not the same as the so-called "Arabic" numerals 0 to 9—invented by the Hindus of India before being brought to the West).

If you were required to localize a replacement argument you have two options:

- Hard-code the variable into an error message. This is best if you do not anticipate reuse of the replacement argument in other error messages.
- If reuse of the replacement argument is necessary, declare it as a key/value pair in your `Application.properties` file. Then use `<arg>` with the `key` attribute corresponding to this key and leave out the `resource` attribute, or set `resource="true"`.

The validation of the `userid` field only succeeds if all the declared validators for it pass. The validation for the `password2` field used the `validwhen` validator, which I will describe in more detail in a later section ("Using validwhen").

Validating Nested and Indexed Properties

`userid`, `password`, and `password2` are all *simple* properties (see Chapter 10 for a definition). You can also validate *nested* and *indexed* properties.

For nested properties, simply set the field's `property` attribute using the usual `.` notation to access a nested property. For example, setting `property="contact.email"` validates the `email` property on the `contact` property of the given form.

Validating indexed properties is a little different. Suppose you have a function that returns an array or `Collection` of values, and you want to validate each item on the array. The way to do this is to use the `indexedListProperty` attribute of `<field>`. For example:

```
<field indexedListProperty="items" ...
```

interprets `items` as an array or `Collection`, and will validate each element in the array/`Collection`.

You can also combine indexed with simple or nested properties. For example:

```
<field indexedListProperty="items" property="id" ...
```

will validate the `id` property of each element on the `items` array/`Collection`. And

```
<field indexedListProperty="contacts" property="address.postcode" ...
```

will validate the nested property `address.postcode` on each element in the `contacts` array/`Collection`.

Using Constants

Both `userid` and `password` validations check that the values keyed in by the user are composed of alphanumerics. In Listing 15-5, the regular expression for each would be duplicated. Is there a way to "share" these? The answer is "yes," and the way to do this is to declare the regular expression as a constant. You have two options open to you:

- Define a global constant—a constant accessible to every field in every `<formset>`.
- Define a constant accessible to every field within a single `<formset>`.

To declare a global constant, you'd place the following XML *before* the first `<formset>` tag:

```
<global>
   <constant>
      <constant-name>alphanumericMask</constant-name>
      <constant-value>^[a-zA-Z0-9]*$</constant-value>
   </constant>
</global>
```

This declares a constant named `alphanumericMask` with the value `^[a-zA-Z0-9]*$`. Declaring the constant in a `<formset>` is similar:

```
<formset>

   <constant>
      <constant-name>alphanumericMask</constant-name>
      <constant-value>^[a-zA-Z0-9]*$</constant-value>
   </constant>

   <form name="RegistrationForm" ...
```

Note that according to the DTD, `<constant>`s have to be declared at the start of the `<formset>` tag.

To use the declared `alphanumericMask` constant in either case, you'd employ the `${constant-name}` reference:

```
<var>
   <var-name>mask</var-name>
   <var-value>${alphanumericMask}</var-value>
</var>
```

Client-Side Validations

To create client-side validations, simply place a `<html:javascript form="MyFormBean"/>` tag anywhere on your JSP page so that Struts can paste JavaScript on the final web page in order to run client-side validations. Bear in mind, however, that there's no easy way (apart from amending the validator itself, or perhaps writing a plug-in) to prevent a form from being passed through server-side validations.

This might make you wonder whether the client-side validations are of any use! To be fair, the Validator framework is undergoing active development, and I'm sure we'll see improvements in the area of client-side validations soon.

We next look at the standard validators available on the Validator framework.

The Standard Validators

Table 15-1 lists all standard validators at the time of this writing (Struts 1.2.4 to 1.2.7). However, as I mentioned the Validator framework is undergoing active development, so be sure to check the Struts Validator website (see the "Useful Links" section) for the latest.

The first column of Table 15-1 shows the name of the validator. This is the name you'd use in the `depends` attribute of the `<field>` tag. The second column gives the list of variables the validator accepts. All variables listed in the second column are *required*. The last column gives "triggers" for the validator—conditions that if *met* will trigger the validator to fail.

Almost every validator is easy to use, with the possible exception of the `validwhen` validator. We'll tackle this next.

Table 15-1. *Standard Validators, Their Arguments, and Their Triggers*

Validator	Variables	Trigger
required	-	Field only has whitespace.
mask	mask	Field doesn't match mask. The mask variable is a regular expression.
intRange,floatRange	min,max	Field > max or field < min.
maxLength	maxLength	Field has more than maxLength characters.
minLength	minLength	Field has fewer than minLength characters.
double,float,long,integer,short,byte		Field can't be cast to associated primitive.
date	datePattern or datePatternStrict	Field doesn't match given date pattern. Note: datePatternStrict requires any leading zeroes present in the pattern to be present in the field.
creditCard		Field is not a credit card number.
email		Field is not a conformant email address. (Note: This is actually much more complicated that testing for a single @ in the field!)
url	schemes, allowallschemes, allow2slashes, nofragments (all optional)	Field is not a URL. Several protocols are supported. schemes is a comma-delimited list of protocols, for example, http,https, ftp,file,telnet,gopher. allowallschemes (which equals false by default) makes any scheme acceptable if true. allow2slashes (= false) accepts double backslashes (//) in a scheme. nofragments (= false) *disallows* a fragment of a URL.
validwhen	test	The test condition fails. See the following subsection for more details.
requiredif	Deprecated	Deprecated—use validwhen instead.
range	Deprecated	Deprecated—use intRange *or* floatRange instead.

Using validwhen

The validwhen validator is useful when validation of one field depends on another. With this validator, you can formulate complex dependencies.

I've always believed an example beats a thousand descriptions, so Listing 15-6 shows an example of validwhen in action.

Listing 15-6. *Using validwhen*

```
<field property="mySecondField" depends="validwhen">
   <var>
      <var-name>test</var-name>
      <var-value>
         ((myFirstField == null) or (*this* != null))
      </var-value>
   </var>
</field>
```

mySecondField is valid when myFirstField is blank or mySecondField is not blank. The symbol *this* refers to the field being validated. Notice also that the logical operators are or and and, *not* the usual Java || and &&.

■ **Note** This is an example of how to conditionally validate fields. mySecondField only gets validated (checked for a blank value) if myFirstField is blank. This is how you can use validwhen to replace the older (pre-1.2) validator requiredif.

Clearly validwhen is a powerful solution to a variety of problems. Consider the common problem of validating a twice-entered password (see Listing 15-7).

Listing 15-7. *Checking for Identical Fields*

```
<field property="password2" depends="validwhen">
   <var>
      <var-name>test</var-name>
      <var-value>
         (password == *this*)
      </var-value>
   </var>
</field>
```

Using validwhen with Indexed Fields

validwhen also makes validating dependent indexed fields and their properties unbelievably easy.

For example, suppose your form had a table of values, such as a list of items in a shopping cart. You could do this by using one of the following:

- Separate arrays for each column
- A single array of containing objects that holds values for a single row

We'll tackle these two situations separately.

In the first scenario, our shopping cart has two columns, each represented by arrays `itemID` and `itemCount`. We want to check that each non-null `itemID` element has a corresponding non-null `itemCount` element. The validator declaration to do this appears in Listing 15-8.

Listing 15-8. *Validating Indexed Fields*

```
<field property="itemCount" depends="validwhen">
   <var>
      <var-name>test</var-name>
      <var-value>
         ((itemID[] == null) or (*this* != null))
      </var-value>
   </var>
</field>
```

The validator automatically runs the check for each row of `itemCount` and `itemID`.

In the second scenario, we use an object called `item` to hold a single row of data. This object has the properties `id` and `count`.

■ **Note** Recall that for this to work, `item` needs to be a JavaBean. That is, it needs to have `getXXX` and `setXXX` functions for each property, like `id` and `count`.

Listing 15-9 shows how we run the same validation.

Listing 15-9. *Validating Properties on Indexed Fields*

```
<field property="count"
       indexedListProperty="item" depends="validwhen">
   <var>
      <var-name>test</var-name>
      <var-value>
         ((item.id[] == null) or (*this* != null))
      </var-value>
   </var>
</field>
```

Lastly, note that you can explicitly refer to a specific row on an indexed field. Listing 15-10 shows an example.

Listing 15-10. *Referring to an Explicit Row on an Indexed Field*

```
<field property="myField1" depends="validwhen">
   <var>
      <var-name>test</var-name>
      <var-value>
         ((item.id[1] == null) or (*this* != null))
      </var-value>
   </var>
</field>
```

In the next section, we'll look at how to run your own validations alongside the standard ones you've used in `validation.xml`.

Adding Custom Validations

Although the set of validations on the Validator framework are quite comprehensive, you might want to run your own validations alongside those available in the Validator framework. There are two ways of doing this:

- Put in a `validate()` function on your `ValidatorForm` subclass. You'd take this approach if the custom validation isn't likely to be reused in other parts of your application.

- Extend the Validator framework itself by creating a new validator. You'd take this route if there's a strong case for reuse.

In most cases, putting in a `validate()` function should suffice. Use the second alternative only after much thought, since validators should be *generic*, if only to the problem domain of your web application. Otherwise, you'd run into potential maintenance issues. Can you see why?

We'll take a look at both approaches in this section.

Implementing validate()

As I mentioned earlier, one way of implementing your own validations in addition to the standard ones is to implement the `validate()` function.

We'll take up the `RegistrationForm` class again (see Listing 15-4), and add custom validations to detect that the given password isn't in a list of common words like "password"

or "secret." Listing 15-11 shows the `validate()` function that you'd add to `RegistrationForm` in order to achieve this functionality.

Listing 15-11. *Checking for Common Words as Passwords*

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request){

   //run the validations declared in validation.xml first
   ActionErrors errors = super.validate(mapping,request);

   if(errors != null && !errors.isEmpty()) return errors;

   errors = (errors == null)? new ActionErrors() : errors;

   //run your custom validations next
   for(int i = commonWords.length - 1; i >= 0; i--){
      if (_password.toUpper().equals(commonWords[i])){
         errors.add("password",
                     new ActionMessage("reg.error.password.toocommon",
                                       commonWords[i]));
         return errors;
      }
   }

   return (errors.isEmpty())? null : errors;
}
```

In Listing 15-11, `validate()` is called on the base class (`ValidatorForm`), which runs the validations defined in `validation.xml`. The end result is an `ActionErrors` object, to which we can add new error messages of our own. Notice the use of the second constructor of `ActionMessage` (see Appendix B), in order to customize the error message with the common word that was rejected in the validation.

Extending the Validator Framework

Instead of implementing `validate()` as in Listing 15-11, we could extend the Validator framework. Do this sparingly. For the Registration webapp example, it would be overkill—implementing `validate()` would suffice. However, in this subsection, I will extend the Validator framework for the same functionality, simply so you can compare the two approaches.

So, in what situations would extending the Validator framework be appropriate? Actually, only when you can justify reuse of the custom validator. This usually means that the custom validator checks a variable prevalent in your webapp's problem domain.

Good examples are *identifiers*, like product codes, ISBN numbers, IP addresses, or zip codes (or "postal codes" as they are known outside the United States). While these can be validated using the standard `mask` validator, it may not always be easy to do so, especially if you need to take care of exceptions to the rule (e.g., postal codes), or if the variable can change format over time (e.g., IP addresses).

You'd be much better off extending the Validator framework than implementing the `validate()` function because you obviously want to *reuse* the validations. A zip code field, for example, could appear in more than one form, or be present in many different webapps you might create.

Creating a new validator is easy:

1. Implement the Java handler class that actually performs the validation.
2. Optionally create JavaScript for form validation. I won't cover this topic in this book.
3. Declare the new handler in `validator-rules.xml` with the `<validator>` tag.

Implementing the Java Handler

This is the trickiest part of the whole process of creating a custom validator. Most of the complexity arises because you must handle the possibility of nested and indexed properties and because you have to use helper classes (`Resources`, `GenericValidator`, etc.) to read the information you need to validate an input datum or to reuse validations available in the Validator framework.

Your Java handler can be *any* Java object, as long as the following conditions are met:

- It implements `java.io.Serializable`.
- The function performing the custom validation has `public` access. It must also be a `static` function that returns a `boolean`, which is `false` if validation fails and `true` otherwise.

You can call this function anything you like. The standard method signature is shown in Listing 15-12.

Listing 15-12. *Standard Signature for the Custom Validation Function*

```
public static boolean
myValidationFunction(Object bean, ValidatorAction va,
                     Field field, ActionMessages errors,
                     Validator validator, HttpServletRequest request){
```

The function's six arguments are the following parameters:

- `java.lang.Object`: This can be a `String` or a JavaBean-conformant object. It holds the value(s) to be validated. If it's a JavaBean, the value to be validated resides on one of the bean's properties. Your validation function will have to handle each of these possibilities.

- `org.apache.commons.validator.ValidatorAction`: This class contains information declared within the `<validator>`. You'll probably never call functions on this class, but an initialized instance of it is needed when you want to load pooled error messages saved in Struts' `Resource` class. Without such an instance, you'd have to resort to calling a new `ActionMessage(...)` each time you wanted to create an error message, which isn't efficient.

- `org.apache.commons.validator.Field`: This contains data pertaining to the field being validated. With this instance, you can find out the name of the property being validated and any other information contained in the `<field>` tag.

- `ActionMessages`: This is simply a container for you to place your error messages. Struts manages the creation of `ActionMessages` objects, enabling these objects to be pooled for efficiency.

- `org.apache.commons.validator.Validator`: This is an instance of the Validator framework itself, which you can use to read other fields in the `<form>`. You might need to use this to perform field-dependent validations (as is the case with `validwhen`). Like `ValidatorAction`, an instance is needed to load pooled `ActionMessage` instances for efficiency.

- `HttpServletRequest`: This is the request object associated with the current request.

■ **Note** You can actually use *any subset* of these six parameters, *in any order*! The Validator framework will resolve the correct function at runtime, based on your declaration in the `<validator>` tag for this validation. However, keeping the function signature consistent makes maintenance that much easier.

A typical validation function will run through these steps:

1. Determine the value to be validated. You will use the helper class `org.apache.commons.validator.ValidatorUtils` to do this.

2. Collect information about the environment (such as the values of other fields on the form), if necessary. You might do this with the `Field`, `ValidatorAction`, and `Validator` instances sent in the function's argument list.

3. Run the custom validation. Knowledge of the functions on `org.apache.commons.validator.GenericValidator` is helpful, since it contains many useful validations you can reuse.

4. Locate a suitable `ActionMessage` instance (instead of calling `new`), and place this instance on the given `ActionMessages` object. Use the helper class, `org.apache.struts.validator.Resources`, to do this.

Listing 15-13 shows a simple validation—ensuring the given data is alphanumeric.

Listing 15-13. *Java Handler to Validate Alphanumeric Strings*

```
package com.mycompany.myapp.struts.validator;

import java.io.Serializable;

import org.apache.commons.validator.ValidatorAction;
import org.apache.commons.validator.Field;
import org.apache.commons.validator.Validator;
import org.apache.struts.action.ActionMessages;
import javax.servlet.http.HttpServletRequest;

import org.apache.commons.validator.util.ValidatorUtils;
import org.apache.commons.validator.GenericValidator;
import org.apache.struts.validator.Resources;

public class MyValidations implements Serializable{

   public static boolean isAlphanumeric(Object bean,
                                        ValidatorAction va,
                                        Field field,
                                        ActionMessages errors,
                                        Validator validator,
                                        HttpServletRequest request){
```

```
        //step 1: Determine value
        String value = null;
        if(String.class.isInstance(bean)){
           value = (String) bean;
        }else{
           value = ValidatorUtils.getValueAsString(bean,
                                                   field.getProperty());
        }

        //step 2: Collect info about the environment
        /* --------       not needed      --------- */


        //step 3: Run the custom validation
        try{
          if(!(GenericValidator.isBlankOrNull(value) ||
               GenericValidator.matchRegexp(value, "^[a-zA-Z0-9]$"))){

               errors.add(field.getKey(),
                     /**
                      * step 4: Locate a suitable ActionMessage
                      *         instance.
                      */
                     Resources.getActionMessage(validator,
                                                request,
                                                va,
                                                field));

               return false;
            }

          }catch(Exception ignore){}

          return true;
      }
}
```

A systematic and detailed exposition of the Apache Commons Validator classes is beyond the scope of this book. In most cases, however, the following information will prove useful:

- `field.getProperty()` gives the property of the field being validated.
- `field.getKey()` gives the error message associated with this validator.
- `ValidatorUtils.getValueAsString(bean, property)` reads the simple/nested/indexed property of the given bean.
- `GenericValidator.isBlankOrNull(value)` returns `true` if the value is blank or `null`.
- `GenericValidator` has many other functions you might find useful, like `matchRegexp(value,regexp)`, which returns `true` if the value matches the given regular expression.
- The Struts class `org.apache.struts.validator.FieldChecks` is the Java handler class corresponding to every one of the Validator framework standard validators.

Armed with the information from this section, you should be well equipped to pursue the topic further by reading the Commons Validator source code, as well as `FieldChecks` from Struts.

■ **Note** To compile your code, you will need `commons-validator.jar`, `commons-beanutils.jar`, `servlet-apis.jar`, and `struts.jar` on your classpath. These files are in the Struts distribution.

The last step is to declare your validator in `validator-rules.xml`. Listing 15-14 shows how this would look in the case of alphanumeric validation.

Listing 15-14. *Declaring the Alphanumeric Validator*

```
<form-validation>
   <global>
      <validator name="alphanumeric"
        classname="com.mycompany.myapp.struts.validator.MyValidations"
        method="isAlphanumeric"
        methodParams="java.lang.Object,
                      org.apache.commons.validator.ValidatorAction,
                      org.apache.commons.validator.Field,
                      org.apache.struts.action.ActionMessages,
                      javax.servlet.http.HttpServletRequest"
        msg="errors.alphanumeric"/>

      // ... other <validator> declarations
```

The name of the validator is what it will be called in your validation.xml file. The classname attribute corresponds to the class containing the validation function. The method attribute is the actual method used for this validator. The methodParams attribute declares the method signature of the validation function.

The msg parameter defines a *default* error message key that the user has to implement in the Application.properties file. It is preferable, however, to override this default message key and specify your own, as I've done in Listing 15-4.

You may add a depends attribute to your <validator> declaration. This functions just like depends on your <field> tags: the comma-delimited list of validators are fired before your custom validation is called. For example, in Listing 15-15 I've amended Listing 15-14 slightly.

Listing 15-15. *Using depends in the Alphanumeric Validator*

```
<validator name="alphanumeric"
      classname="com.mycompany.myapp.struts.validator.MyValidations"
      method="isAlphanumeric"
      methodParams="java.lang.Object,
                    org.apache.commons.validator.ValidatorAction,
                    org.apache.commons.validator.Field,
                    org.apache.struts.action.ActionMessages,
                    javax.servlet.http.HttpServletRequest"
      msg="errors.alphanumeric"
      depends="required"/>

      // ... other <validator> declarations
```

The amended validator of Listing 15-15 would fail if the user entered a blank value, since it depends on the required validator.

Migrating Legacy Code

As your webapp grows, creating and maintaining individual validate() functions containing mainly generic validations can be a challenge. As I've discussed in the introduction to this chapter, using the Validator framework brings many benefits, so migrating your legacy code to the Validator framework might be something you'd want to do. Fortunately, this is a simple three-step process:

1. Change the base class from ActionForm to ValidatorForm.

2. Comment out the old validate() function on the legacy code. This makes it easy for you to "roll back" to the old validations if you detect a bug in the new validations.

3. As usual, declare the required validations in `validation.xml`. If not all validations can be moved to the Validator framework, use the two techniques described earlier to accommodate these special cases.

It's wisest to do migrate incrementally, and use a unit-testing framework (see "Useful Links") to test the changes.

Localizing Validations

In some situations, you might need to localize your validations. It's rare to want to do this, and in most scenarios where you might want to do so (such as checking for zip codes), the validation can be misleading. I've discussed the pitfalls of localizing validations in Chapter 12, and given you a couple of useful hacks to help you do this if you need to.

The Validator framework does provide support for localizing validations, but the solution is a poor one. I should hasten to add that this isn't a great failing since you should think twice (or three times!) before deciding to localize any validation.

■ **Caution** Do not confuse localizing validations with localizing output (e.g., error messages). The Validator framework provides good support for localizing error messages through the use of keys/value pairs on `Application.properties`.

Recall that the `<formset>` tag is a containing tag for all your declared validations. In order to create validations specific to a locale, you'd have to put in a new `<formset>` in the root containing the `<form-validation>` tag.

This new `<formset>` tag would contain *all* necessary validations for the new locale (refer to Chapter 12 for a discussion on locales). Notice the emphasis on "all"—there's no way for you to selectively override the validations for just a few fields; you'd have to specify the validations for *every* field again in the new `<formset>`.

The `<formset>` tag accepts three optional properties that you can use to specify a locale:

- `language`: This is a ISO-639 identifier for a language. Examples are `en` for English or `zh` for Chinese.
- `country`: This is an ISO-3166 identifier for a country. For example, the code `US` refers to the United States.
- `variant`: This is a variant on a given language/country combination. The user usually can't set this within the web browser.

If all these attributes are left unspecified, the `<formset>` element is taken to be the default one.

The big disadvantage of using <formset> for localizing validations is that you can't "override" the validations of just a few fields. So, if you really, really need to localize your validations, I'd recommend either one of the two hacks I presented in Chapter 12.

Lab 15: Using the Validator Framework in LILLDEP

In Lab 6, you hand-coded various validations for ContactForm. In this lab session, you'll migrate ContactForm to using the Validator framework.

1. Put in the plug-in section for the Validator in struts-config.xml. Use the name default name validation.xml for your validation set.

2. Change the base class of ContactForm to org.apache.struts.validator.ValidatorForm.

3. Comment out the existing validate() function.

4. Complete validation.xml in ./web/WEB-INF. Declare all the validations you implemented in Lab 6. If you completed Lab 10b, you would also need to amend the relevant JSPs.

5. Compile, deploy, and test your changes.

Useful Links

- RFC-2822 for email: http://rfc.net/rfc2822.html
- Struts' Validator page: http://struts.apache.org/struts-doc-1.2.x/userGuide/dev_validator.html
- The Commons Validator framework project: http://jakarta.apache.org/commons/validator/
- A Regular Expressions tutorial: http://www.regular-expressions.info/tutorial.html
- Some information on unit testing (also the homepage of JUnit, the Java unit testing framework): www.junit.org

- ISO-639 language codes: `http://en.wikipedia.org/wiki/ISO_639`
- ISO 3166 country codes: `http://en.wikipedia.org/wiki/ISO_3166`

Summary

- The Validator framework brings many benefits by simplifying the creation and maintenance of your webapp's validations.
- You may extend the Validator framework either by implementing your own `validate()` function or by creating a new validator.

CHAPTER 16

Dynamic Forms

Every time you need a new form for user input, you usually have to write a new `ActionForm` subclass. This can get tedious quickly, especially if your form has many getter and setter functions. (I hope you had a taste of how bad it can get in Lab 6, where you had to implement `ContactActionForm`, with 14 getters and setters.)

In previous chapters, I've shown you how to alleviate these problems a little by reusing forms (Lab 9a) or by using nested properties (Lab 10a) to avoid writing superfluous getter and setter functions in your `ActionForm` subclasses.

But these get you only so far. *Dynamic forms*, the subject of this chapter, are a way to eliminate writing Java code entirely. Instead, you declare additional XML in `struts-config.xml` for each form you want to create. The XML involved is an extension to the usual `<form-bean>` declaration you're familiar with: you add `<form-property>` tags for each property in the form.

Like everything else in life, there are advantages as well as disadvantages to using dynamic forms. The primary advantage is, of course, that you don't have to declare getters and setters for your form. This can be quite a big plus if your application requires a lot of forms. However, as you'll see later, dynamic forms also have a number of important shortcomings.

Toward the end of this chapter, we'll also look at a couple of recent additions to Struts (since Struts 1.2.6) that take the dynamic forms idea to the extreme. Using `LazyValidatorForm`, you don't even have to declare the properties in your forms!

We won't be done with dynamic forms in this chapter. In Chapter 19, I'll show you how to implement a plug-in that makes interesting extensions to dynamic forms.

Declaring Dynamic Forms

To create a dynamic form, you need only to declare it in your `struts-config.xml` file. Declaring a dynamic form is easy: in your `<form-bean>` declaration of the form, put in extra `<form-property>` tags for each property on the form.

You can declare *simple*, *indexed*, *mapped*, or *nested* properties this way (see Chapter 10).

Listing 16-1 shows how you might declare a form bean with a simple (`String`), indexed (array), and mapped property (using a `HashMap`).

Listing 16-1. *Declaring a Dynamic Form*

```
<form-bean name="MyFormBean"
          type="org.apache.struts.action.DynaActionForm">

   <!-- declaring a simple property -->
   <form-property name="MySimpleProp"
                  type="java.lang.String"
                  initial="Hello" />

   <!-- declaring another simple property -->
   <form-property name="MySimpleIntProp"
                  type="int" />

   <!-- declaring an indexed property -->
   <form-property name="MyIndexedProp"
                  type="java.lang.String[]"
                  size="314" />

   <!-- declaring a mapped property -->
   <form-property name="MyMappedProp"
                  type="java.util.HashMap" />

   <!-- declaring a nested property -->
   <form-property name="MyNestedProp"
                  type="com.myco.myapp.MyBean" />

</form-bean>
```

The first thing to note is that the `type` attribute *must* be either `org.apache.struts.action.DynaActionForm` (itself a subclass of `ActionForm`) or your subclass of `DynaActionForm`.

To put it another way, Struts disregards a `<form-bean>`'s `<form-property>` tags *unless* the `ActionForm` subclass indicated by the `type` attribute is also a subclass of `DynaActionForm`. For Struts versions later than 1.2.6, this is not strictly true. See the note for details.

■ **Note** In versions of Struts later than 1.2.6, if the class indicated by the `type` attribute is of type `BeanValidatorForm` or a subclass of it (like `LazyValidatorForm`), then the `<form-property>` tags are read as well. I'll describe both of these interesting new additions toward the end of this chapter.

Declaring Simple Properties

From Listing 16-1, you can easily see that simple properties can either be supported Java classes or primitive types. Java classes and primitive types supported by `DynaActionForm` are

- `java.lang.BigDecimal`
- `java.lang.BigInteger`
- `boolean` and `java.lang.Boolean`
- `byte` and `java.lang.Byte`
- `char` and `java.lang.Character`
- `java.lang.Class`
- `double` and `java.lang.Double`
- `float` and `java.lang.Float`
- `int` and `java.lang.Integer`
- `long` and `java.lang.Long`
- `short` and `java.lang.Short`
- `java.lang.String`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`

(This list is available on the Struts documentation site. Refer to "Useful Links" at the end of this chapter.) You may optionally specify a default value for the simple property using the `initial` attribute. For example, in Listing 16-1, the default value for the simple property `MySimpleProp` is `Hello`.

Declaring Indexed Properties

To create an indexed property, simply put `[]` after the classname (or primitive type). You also need to specify a size for the array. If you need *flexible* length arrays, there are two ways to do it.

In the first method, you omit the `size` attribute. Since the size isn't specified, a newly created form bean will have a `null` value for this indexed property. So, you have to "prime" the form bean (a `DynaActionForm` instance) in an `Action` subclass before it can be used to store data. You've seen a variation of this technique in Chapter 10.

When a page is requested, it's really a request to a form handler. This form handler (an `Action` subclass) creates an array of the right type and length and inserts it into the form bean. Only then is control passed to a JSP page, which populates the form. `DynaActionForm` has a function called `set(String propertyName,Object value)`, which you can use to prime the form bean. Listing 16-2 illustrates this technique in action.

Listing 16-2. *Setting Array Lengths at Runtime*

```
public ActionForward execute(...){

   DynaActionForm dForm = (DynaActionForm) form;

   /* somehow get the needed array length */
   int length = ...;

   /* create the new array */
   String[] myProp = new String[length];

   /* insert array into form */
   dForm.set("MyIndexedProp", myProp);

   /* forward to JSP */
   return mapping.findForward("success");
}
```

The JSP page pointed to by `success` will be able to use the form's `MyIndexedProp` property. Needless to say, you need to use session scope for this to work.

The second method for flexible arrays is to use a class that implements the `java.util.List` interface (e.g., `ArrayList`) as the `<form-property>`'s `type`:

```
<form-property name="MyIndexedProperty" type="java.util.ArrayList" />
```

Unfortunately, with this approach you lose Struts's automatic type conversion, since all data stored in the `List` are stored as `String` instances.

Declaring Mapped Properties

Mapped properties are defined by using a `type` that implements the `java.util.Map` interface (e.g., `HashMap`):

```
<form-property name="MyMappedProperty" type="java.util.HashMap" />
```

Declaring Nested Properties

Lastly, nested properties are defined using a `type` attribute, which is a JavaBean, assumed to contain nested properties used in your JSPs:

```
<form-property name="MyNestedProperty" type="com.myco.myapp.MyBean" />
```

The only thing to note about nested properties is that you must ensure that the nested property finally resolves to one of the supported primitive types or Java classes listed earlier for simple properties. This should hardly come as a surprise, since this is what you'd expect even with ordinary `ActionForms`.

Accessing Dynamic Properties

At some point, you will need to read (or write) data from your dynamic form. The `DynaActionForm` base class exposes three getters corresponding to simple, indexed, and mapped properties:

- `get(String property)` returns an `Object` value of the given property. This accessor is for simple properties.
- `get(String property,int index)` returns an `Object` value of the given property with the given index. This accessor is for indexed properties.
- `get(String property,String key)` returns an `Object` value for the given key on the given property. This accessor is for mapped properties.

Similarly, there are `set(String property, Object value)`, `set(String property, int index, Object value)`, and `set(String property, String key, Object value)`.

If you've used a primitive type for a property, then the `Object` type in the return of `get()` and the argument of `set()` would be the corresponding Java class for that primitive type. For example, if you've declared a property of type `int`, then you should expect an `Integer` to be returned from `get()` and you should pass an instance of `Integer` in `set()`.

Transferring Properties to a JavaBean

In many instances, all you want to do with a dynamic form bean is to transfer the property values into a JavaBean instance. This is often the case if:

- You're using Struts as a web-based front-end to an existing application. The existing application usually would already have JavaBean classes to hold data. You want to transfer properties from your dynamic form to this JavaBean.

- You want to transfer data into a data object. This is the case if you're using a Model framework like Torque or Lisptorq, which exposes data through data objects. See Chapter 5 and Appendix A for details.

The Apache Commons' Bean project (see "Useful Links") provides a utility class to help you to do this transfer easily. The following

```
org.apache.commons.bean.BeanUtils.copyProperties(myJavaBean , form)
```

will copy the properties of the `DynaActionForm` instance (`form`) into your JavaBean class (`myJavaBean`).

Of course, only properties existing on `myJavaBean` will be copied over. The JAR file for Apache Bean classes are bundled with Struts in `commons-beanutils.jar`.

Dynamic Form Disadvantages

Dynamic forms have a few significant limitations compared to ordinary `ActionForms`:

- **You must use different syntax properties when using EL**: If you're using EL, you have to prefix `map` to a form's property. For example, instead of using `${myform.myproperty}`, you'd have to use `${myform.map.myproperty}`. This applies to simple, indexed, mapped, and nested properties. This is only necessary when referring to a property using EL. No change would be necessary when referring to a property within either the original or EL-enabled Struts tags. For example, `<bean:write name="myform" property="myproperty"/>` would work fine.

- **You can't use conventional getters and setters**: If you subclass `DynaActionForm` (in order to run custom validations, for example), you can't use conventional getters and setters on your subclass and expect them to work with Struts tags. Struts handles `DynaActionForm` and its subclass differently from normal `ActionForms`. This can be a serious problem if you need to work directly with form data. The design philosophy of dynamic forms assumes that you *won't* have to work with their data directly. Rather, you are expected to transfer the form's properties to a suitable JavaBean, using the `BeanUtils` technique described in the previous subsection.

- **Compile-time checking is not possible**: By using dynamic forms, you lose compile-time checking. For example, because the return value of `DynaActionForm`'s `get()` are all `Objects`, there's no way to tell *at compile time* if you've assigned a return value to the wrong type. This would not be a problem with conventional getters because their return types are usually quite narrow (`String` or `Double`, etc.). A mismatch in assignment would be caught by the Java compiler. A similar situation occurs for setters as well.

When to Use Dynamic Forms

So, when *do* you use dynamic forms? The Apache Struts User's Guide (see "Useful Links") spells this out in detail:

> *DynaActionForms are meant as an easy solution to a common problem: Your ActionForms use simple properties and standard validations, and you just pass these properties over to another JavaBean...DynaActionForms are not a drop-in replacement for ActionForms...*

The first sentence refers to the use of `BeanUtils` to transfer properties over to a JavaBean. You've seen this in action in the previous subsection.

Using "standard validations" means using the Validator framework. This is *recommended* because frequent accessing of form properties requires you to use `DynaActionForm`'s `get()` functions, which is an error prone approach since you've lost compile-time checking of your code. The `get()` function isn't error prone, but your code might have bugs and there's no way to find this out at compile time. This problem just doesn't exist when you use conventional `ActionForms`, since the compiler does the check for you.

The second sentence refers to the fact that a dynamic form's properties have a different EL reference than the conventional `ActionForm`'s properties. This can be a brick wall if you're trying to replace regular `ActionForms` in an application that makes heavy use of EL.

Validating Dynamic Forms

Like ordinary `ActionForms`, there are two ways to validate data in a dynamic form:

- Implement a subclass of `DynaActionForm` with a `validate()` function.
- Use a subclass of `DynaActionForm` called `org.apache.struts.validator.DynaValidatorForm`, which uses the Validator framework. (See Chapter 15 for details.)

As with `ActionForms` using the Validator framework, you can run your own custom validations on top of the `DynaValidatorForm` by simply subclassing it and calling `super.validate()` in your subclass's `validate()` function. I've described this technique in Chapter 15.

Used together with the Validator framework and `BeanUtils`, dynamic forms can completely eliminate the need to write any Java code for a form bean. But before we go into this, I'll show you how the Registration webapp (see Chapter 5) might be implemented using dynamic forms.

The Registration Webapp with Dynamic Forms

In Chapter 5, I introduced the simple Registration webapp, which I expanded on in Chapter 14. In this section, I'll use a simple variation on the Registration webapp to show you how you might use dynamic forms.

The Registration webapp consists of a single page. This page has one form containing three fields: the user ID, the password, and the password retyped, as shown in Figure 16-1.

Register New User - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/registratior
Register New User
Please enter a userid (between 5 to 15 characters):
Please enter a password (between 8 to 20 characters):
Please re-enter the password:
Submit
Reset
Done

Figure 16-1. *The Registration webapp main page*

In Listing 16-3, I've reproduced the JSP code from Chapter 8 for your convenience.

Listing 16-3. *registration.jsp*

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html>
<head>
   <title><bean:message key="registration.jsp.title"/></title>
</head>
<body>
      <h1><bean:message key="registration.jsp.heading"/></h1>
      <html:form action="Registration.do" focus="userid">
         <p>
           <bean:message key="registration.jsp.prompt.userid"/>
```

```
            <html:text property="userid" size="20" />
            <html:errors property="userid" />
          </p><p>
            <bean:message key="registration.jsp.prompt.password"/>
            <html:password property="password" size="20" />
            <html:errors property="password" />
          </p><p>
            <bean:message key="registration.jsp.prompt.password2"/>
            <html:password property="password2" size="20" />
          </p>

          <html:submit>
             <bean:message key="registration.jsp.prompt.submit"/>
          </html:submit>

          <html:reset>
             <bean:message key="registration.jsp.prompt.reset"/>
          </html:reset>
       </html:form>
</body>
</html:html>
```

Notice that this is a straight copy of Listing 8-1. I didn't have to make any changes to the JSP in order to use dynamic forms. As I've discussed previously, this is because I haven't used EL in the tags.

Previously, in Chapter 6, I had to use an `ActionForm` subclass called `RegistrationForm` (see Listing 6-1) in order to store and validate the form data. No longer! Since I'll use `DynaValidatorForm` and the Validator framework, I no longer need this class.

Listing 16-4 shows how to declare the dynamic form (the changes from Listing 6-1 are in bold font).

Listing 16-4. *struts-config.xml for the New Registration Webapp*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
   "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>

   <form-beans>
     <form-bean
        name="RegistrationForm"
        type="org.apache.struts.validator.DynaValidatorForm">
```

```
        <form-property name="userId"    type="java.lang.String" />
        <form-property name="password"  type="java.lang.String" />
        <form-property name="password2" type="java.lang.String" />

      </form-bean>
    </form-beans>

    <global-exceptions>
      <exception key="reg.error.io-unknown"
                 type="java.io.IOException"
                 handler="net.thinksquared.registration.ErrorHandler"/>

      <exception key="reg.error.unknown"
                 type="java.lang.Exception"
                 path="/errors.jsp" />
    </global-exceptions>

    <global-forwards>
      <forward name="ioError" path="/errors.jsp"/>
    </global-forwards>

    <action-mappings>
      <action
         path="/Registration"
         type="net.thinksquared.registration.struts.RegistrationAction"
         name="RegistrationForm"
         scope="request"
         validate="true"
         input="/Registration.jsp">

          <forward name="success" path="/Success.jsp"/>

      </action>
    </action-mappings>

    <message-resources parameter="Application"/>

    <plug-in className="org.apache.struts.validator.ValidatorPlugIn" >
      <set-property property="pathnames"
        value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
    </plug-in>

</struts-config>
```

The declarations in validations.xml to perform simple validations are exactly as you've just seen in Chapter 15 (see Listing 15-5).

I'll also have to make changes to the RegistrationAction (see Chapter 7, Listing 7-3) in order to correctly read data from the ActionForm passed into execute(). I've bolded the changes in Listing 16-5.

Listing 16-5. *The New RegistrationAction.java*

```
package net.thinksquared.registration.struts;

import javax.servlet.http.*;
import org.apache.struts.action.*;
import net.thinksquared.registration.data.User;

public class RegistrationAction extends Action{

   public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response){

      //get userid and password (changed!)
      DynaActionForm dForm = (DynaActionForm) form;
      String userid = dForm.get("userid");
      String password = dForm.get("password");

      //complex validation: check if userid exists
      if(User.exists(userid)){
         ActionMessages errors = new ActionMessages();
         errors.add("userid",
                    new ActionMessage("reg.error.userid.exists"));

         saveErrors(request,errors);

         //navigation: redisplay the user form.
         return new ActionForward(mapping.getInput());

      }else{

         //data transformation: save the userid and password
         //to database:
```

```
            User user = new User();
            user.setUserId(userid);
            user.setPassword(password);
            user.save();

            //navigation: display "you're registered" page
            return new ActionForward(mapping.findForward("success"));
        }
    }
}
```

Only the section in bold font was changed. The other portions of `RegistrationAction` are the same as in Listing 7-3.

See Ma, No Hands!: LazyValidatorForm (Struts 1.2.6+)

Version 1.2.6 of Struts saw the introduction of a new type of dynamic form, called `LazyValidatorForm`. As its name suggests, `LazyValidatorForm` uses the Validator framework to run validations. You can subclass `LazyValidatorForm` and write your own `validate()` function if you want to.

The special thing about `LazyValidatorForm` is that you don't have to declare *any* form properties—not for simple, mapped, or indexed properties. For these properties, `LazyValidatorForm` will automatically create the necessary simple, mapped, or indexed property if none exists.

Of course, if you're using a nested property, you'd have to declare it since there's absolutely no way for `LazyValidatorForm` to know which bean class to use for that property. The declaration is the same as for conventional dynamic forms.

However, if you *wish* to declare a property (of any type), you may do so with the usual `<form-property>` tag. There are three reasons why you might want to do this:

- You want to use a **different implementation of** `Map` **or** `List` that `LazyValidatorForm` uses internally. You might do this for reasons of efficiency (your code is faster than the implementations `LazyValidatorForm` uses) or for additional functionality that the default implementations do not have (e.g., logging or debugging).

- You want to take advantage of **automatic type casting**. Data is always stored as `String` instances unless you explicitly demand otherwise. In order for `LazyValidatorForm` to know what type to cast a given property, you have to specify its type using `<form-property>`. By the way, if you specify an array type, `LazyValidatorForm` will automatically grow that array for you when the form is populated. Neat!

- You don't want to pass a `null` `List` to the Validator framework. If no values are passed by the user for an indexed property, that property will have a value of `null`. This will cause the Validator framework to throw an exception. The way around this is to declare the indexed property using `<form-property>`. This tells `LazyValidatorForm` to create a zero-length array or empty `List` of the type you've declared. This keeps the Validator framework happy.

Listing 16-6 shows how Listing 16-1 would look like if you were using `LazyValidatorForm`.

Listing 16-6. *Declaring a Dynamic Form with LazyValidatorForm*

```
<form-bean name="MyFormBean"
          type="org.apache.struts.validator.LazyValidatorForm">

   <!-- declaring a simple property: The same as
        before since the initial value can't be deduced -->
   <form-property name="MySimpleProp"
                  type="java.lang.String"
                  initial="Hello" />

   <!-- declaring another simple property: Required
         since we want automatic type casting -->
   <form-property name="MySimpleIntProp"
                  type="int" />

   <!-- declaring an indexed property: as above, but
        the size attibute is gone. -->
   <form-property name="MyIndexedProp"
                  type="java.lang.String[]" />

   <!-- declaring a mapped property: Gone, because
        MyMappedProperty will be automatically created by
        LazyValidatorForm when the form data is submitted. -->

   <!-- declaring a nested property: The same as before. -->
   <form-property name="MyNestedProp"
                  type="com.myco.myapp.MyBean" />

</form-bean>
```

Now, Listing 16-6 hasn't changed much from Listing 16-1, but that's usually not the case. In many cases, automatic type casting isn't necessary (as it is with LILLDEP's

ContactForm), and neither are initial values. In this case, Listing 16-7 shows how we can pare down Listing 16-6.

Listing 16-7. *Declaring a Dynamic Form with LazyValidatorForm, Take 2*

```
<form-bean name="MyFormBean"
          type="org.apache.struts.validator.LazyValidatorForm">

   <!-- declaring a simple property: Gone! initial
        value not needed -->

   <!-- declaring another simple property: Gone! automatic
        type casting not needed -->

   <!-- declaring an indexed property: Gone! We're sure the
     array will be populated -->

   <!-- declaring a mapped property: Gone! because
        MyMappedProperty will be automatically created by
        LazyValidatorForm when the form data is submitted. -->

   <!-- declaring a nested property: The same as before. -->
   <form-property name="MyNestedProp"
                  type="com.myco.myapp.MyBean" />

</form-bean>
```

As you can see, Listing 16-7 is much better. The declaration for the nested property remains because that's the only way to let LazyValidatorForm know which bean class to use.

Disadvantages of Using LazyValidatorForm

LazyValidatorForm has all the shortcomings of conventional dynamic forms, plus one more: you've lost the "firewalling" property of conventional dynamic forms, as I'll explain shortly.

If form data is submitted to Struts with *more* properties compared to the declared ones (a larger array data or new properties), then Struts throws an exception. This doesn't happen with LazyValidatorForm. This means that a malicious user can craft an HTML page filled with huge amounts of fake "data" and submit this to your webapp, and it will be accepted. The problem may be serious if your code unthinkingly dumps this form data directly into a database. If the data takes time to be processed, your webapp could slow down. In an extreme scenario, your webapp could crash.

This problem doesn't happen with conventional dynamic forms because Struts knows exactly what properties to expect, as well as the length of those properties—except perhaps for `List`- or `Map`-backed properties, but you can avoid using them in the first place. With `LazyValidatorForm`, you don't have the option *not* to use these.

One way to mitigate this problem is to transfer data to JavaBean classes from a `LazyValidatorForm` instance, using the `BeanUtils` technique described earlier. This way, only legitimate data gets processed. Of course, this means that you can't accept mapped or `List`-backed indexed properties, since doing so would allow a malicious user to give you huge lists of fake data.

The Hidden Power of BeanValidatorForm (Struts 1.2.6+)

The base class of `LazyValidatorForm` is `BeanValidatorForm`, which also has interesting uses of its own. This class is a base class of `ValidatorForm`, and therefore uses the Validator framework for simple validation.

The interesting thing about `BeanValidatorForm` is that it accepts a JavaBean in its constructor, automatically populating this JavaBean with the user's form data. Since 1.2.6, if the `type` attribute of the `<form-bean>` isn't a subclass of `ActionForm`, the instantiated object is automatically "wrapped" in an instance of `BeanValidatorForm` by Struts behind the scenes.

To see what this means practically, suppose you were to redo Lab 6 using `BeanValidatorForm` *implicitly*. Instead of creating a `ContactForm` class, you'd first declare the form bean as shown here:

```
<form-bean name="ContactFormBean"
           type="net.thinksquared.lilldep.database.Contact"/>
```

Note that the class referred to by `type` is *not* an `ActionForm` subclass. Struts (since 1.2.6) will automatically instantiate the `Contact` and insert it into the constructor of `BeanValidatorForm`. This *is* an `ActionForm` subclass, and will automatically populate the fields of `Contact`.

To retrieve the `Contact` instance in your `Action`, you simply call

```
BeanValidatorForm bForm = (BeanValidatorForm) form;
Contact contact = (Contact) bForm.getInstance();
```

Easy! Of course, remember that this is only available in 1.2.6 and above.

Lab 16: Deleting Selected Contacts in LILLDEP

In `listing.jsp`, we want to put in check boxes at the left of each contact listing, to allow the user to select contacts for deletion. You will implement this functionality with dynamic forms and `<html:multibox>`.

`<html:multibox>` is usually used with `<logic:iterate>` to render a check box at each iteration, hence the name *multibox*. These check boxes are tied to an indexed property on the form bean. For example:

```
<form-bean name="myForm"
           type="org.apache.struts.action.DynaActionForm">

   <form-property name="myArray" type="java.lang.String[]" />

</form-bean>
```

(This snippet uses a dynamic form, but you could also use a regular `ActionForm`.) The corresponding `<html:multibox>` usage might be

```
<logic:iterate id="aBean" name="myBeans">
 <html:multibox property="myArray"
                   value="<%=((MyBean)aBean).myValue() %>"/>
</logic:iterate>
```

The `value` attribute on the multibox is the actual value submitted by a check box of the multibox. A value is stored *only* if the associated check box is selected.

Also notice that with `<html:multibox>`, there's *no need* to use an explicit indexed property (e.g., `property="myArray[..]"`) in order to populate the form bean.

Armed with this background, complete the following steps in order to implement the deleting functionality on the full listing page.

Step 1: Declare the SelectionForm Form Bean

`SelectionFormBean` is the form bean that will hold the IDs of the `Contacts` selected for deletion. Declare this as a dynamic form. It should have a single array property called `selected`, of type `java.lang.String[]`.

Step 2: Amend listing.jsp

`listing.jsp` needs to display the check boxes to choose `Contacts` for deletion:

1. Put an `<html:form>` tag into `listing.jsp` so that it submits to `DeleteContacts.do`.

2. Put in an `<html:multibox>` tag for input into `SelectionFormBean`. Use a scriptlet if necessary. Extra credit if you use EL-enabled Struts tags to avoid using scriptlets. Hint: The values submitted by the `<html:multibox>` should be the IDs of the `Contacts` to be deleted.

Step 3: Create the Action to Delete Contacts

Complete the implementation of `DeleteContactsAction`, so that it deletes the selected contacts. A couple of hints:

- Use `get(String property, int index)` to read values from the dynamic form.

- This function throws an `IndexOutOfBoundsException` if the index is larger than the available list of items.

When you're done, put in a new action mapping for `DeleteContacts`. `success` should lead back to `Listing.do` (*not* `listing.jsp`!). As usual, compile, deploy, and test your work.

Useful Links

- List of supported types for `DynaActionForm`: `http://struts.apache.org/struts-doc-1.2.x/userGuide/building_controller.html`

- `LazyValidatorForm` JavaDoc: `http://struts.apache.org/struts-doc-1.2.7/api/org/apache/struts/validator/LazyValidatorForm.html`

- `BeanValidatorForm` JavaDoc: `http://struts.apache.org/struts-doc-1.2.7/api/org/apache/struts/validator/BeanValidatorForm.html`

- Apache Commons' `BeanUtils` project: `http://jakarta.apache.org/commons/beanutils/`

Summary

- Dynamic forms are a way for you to create form beans without any Java code.
- Dynamic forms work best when you have simple properties, validated with the Validator framework. You are expected to transfer data to a JavaBean for processing.
- Dynamic forms may not be used in place of `ActionForm`s in certain situations, particularly when using EL.
- `LazyValidatorForm` is a new (1.2.6) addition to Struts that removes the need to even declare properties in a form bean.
- Since 1.2.6, you can use a JavaBean as the `type` of a `<form-bean>`, and Struts will wrap the bean in a `BeanValidatorForm`. This will automatically populate the bean's data for you.

CHAPTER 17

Potpourri

A potpourri (pronounced *poh*-poo-ri) is a mix of spices and flower petals in a cloth bag, valued for its pleasing aroma. This chapter is similarly a mix of useful Struts techniques, which I hope will help your Struts apps "smell" that much better! In order of appearance (and likelihood that you might find them useful), they are

- `PropertyUtils`: A way to read properties on JavaBeans in general, and `ActionForms` in particular, without resorting to type conversions.
- `DownloadAction`: An `Action` subclass that simplifies downloading data to a web client. This data might be dynamically generated by your webapp or a static file on the server.
- `LocaleAction`: Yet another way to allow users to switch locales.
- `IncludeAction` **and** `ForwardAction`: Help you to put a Struts front-end to legacy servlets or JSPs.
- `LookupDispatchAction`: Allows you to have multiple actions on your HTML form *without* using JavaScript.
- `DispatchAction`: A neat way to perform conditional processing in your `Action`, based on the request URL.
- `MappingDispatchAction`: Helps you group related functionality in one `Action`.
- **Global forwards**: A second take on how to use global forwards effectively.
- **Logging**: Struts comes bundled with the Apache Commons Logging, which you can use to perform logging.
- **Wildcards**: A useful trick to help cut down the declarations in a `struts-config.xml` file.
- **Splitting up** `struts-config.xml`: As a webapp grows, the need to split up `struts-config.xml` increases. I'll describe a couple of ways to do this.

PropertyUtils

The Apache Commons Beans subproject has some very useful classes (you've met `BeanUtils` in the previous chapter), among them the `PropertyUtils` class. This class exposes static methods that help you read/write data from/to a JavaBean. The JAR file (`commons-beanutils.jar`) comes bundled with the Struts binaries, so you can use `PropertyUtils` right away in your Struts applications.

No type conversions are used, so you don't have to know the class of the JavaBean at design time. Instead, `PropertyUtils` uses Java's introspection feature to accomplish its job.

■ **Note** `PropertyUtils` actually delegates all its work to `PropertyUtilsBean`. This latter class does the actual work of parsing the property and introspecting the JavaBean. `PropertyUtils` simply provides convenient static access to `PropertyUtilsBean`'s functions (which are not static).

The most useful functions on `PropertyUtils` are

- `boolean isReadable(Object bean, String property)`: Returns `true` if the given property can be read on the bean, `false` otherwise. This implies the existence of an appropriate `getXXX` function.
- `boolean isWritable(Object bean, String property)`: Returns `true` if the given property can be written to the bean, `false` otherwise. This implies the existence of an appropriate `setXXX` function.
- `Class getPropertyType(Object bean, String property)`: Returns the `Class` object associated with the given property.
- `Object getProperty(Object bean, String property)`: Returns the value of the given property.
- `setProperty(Object bean, String name, Object value)`: Writes the given value to the bean.

These functions accept simple, indexed, mapped, nested, or even mixed properties (a combination of the other types of properties). Each of these functions may throw

- `IllegalAccessException`: If the calling code cannot call the associated `setXXX` or `getXXX` function, although it exists
- `NoSuchMethodException`: If the required property does not exist on the bean

- `InvocationTargetException`: If the get/set function on the bean throws an `Exception`
- `IllegalArgumentException`: If either `bean` or `property` is `null`

Using PropertyUtils

This is all very good, but how can `PropertyUtils` be useful in Struts programming? Recall that your `ActionForm` subclasses (including your `DynaActionForm` subclasses) are also JavaBeans. In previous lab sessions, to read (or write) an `ActionForm`'s properties, we had to perform type conversion, as shown in Listing 17-1.

Listing 17-1. *Type Conversion Idiom to Read ActionForm Properties*

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response){

    //get userid and password
    RegistrationForm rForm = (RegistrationForm) form;
    String userid = rForm.getUserId();
    String password = rForm.getPassword();
    ...
```

Listing 17-1 shows an idiom I've used throughout this book. To read the properties in an `ActionForm` subclass, you must do the following:

- Perform a type conversion.
- Read the form's properties.

I've used this idiom primarily for the sake of clarity, since it simplifies the examples and lab session code. You should think twice before using it in production code, if maintainability is important to you. This is because there are two big downsides to using type conversion as in Listing 17-1:

- **No dynamic forms**: You can't easily replace the `ActionForm` with a dynamic form equivalent. You'd have to amend the `Action` code. You've seen this in Chapter 16, when I discussed dynamic forms, and I ported the Registration webapp to using dynamic forms. Refer to Listing 16-5 to see how you'd have to amend Listing 17-1 in order to work with dynamic forms.

- **Tight coupling**: It ties your `Action` class too rigidly to your `ActionForm` class, preventing reuse of the `Action` with other `ActionForms`. For example, if you wanted to reuse the `Action` of Listing 17-1, you'd have to ensure that all the `ActionForms` passed into its `execute()` were of type `RegistrationForm`. In some scenarios, this constraint might be too restrictive.

Instead of this approach, you can use `PropertyUtils` to circumvent both these drawbacks. `PropertyUtils` lets you completely decouple your `Action` from a particular `ActionForm` subclass (this includes dynamic forms).

Using `PropertyUtils`, Listing 17-1 would now look like Listing 17-2.

Listing 17-2. *Using PropertyUtils to Read ActionForm Properties*

```
import org.apache.commons.beanutils.PropertyUtils;
...
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
throws Exception{

      //get userid and password
      String userid = PropertyUtils.getProperty(form,"userid");
      String password = PropertyUtils.getProperty(form,"password");
      ...
```

Using `PropertyUtils` as shown in Listing 17-2, replacing `RegistrationActionForm` with an equivalent dynamic form would be much easier—you wouldn't have to alter your `Action` subclass code.

In a Nutshell...

One downside of using `PropertyUtils` is that like with dynamic beans, you lose compile-time checking. You've essentially traded compile-time checking for looser coupling between classes.

However, in most instances the advantages gained (looser coupling *and* the ability to switch to dynamic forms later) well outweigh this loss.

So, unless you have a really good reason, use `PropertyUtils` to access form or bean properties. Using it consistently makes your code much more maintainable in the long run.

DownloadAction (Struts 1.2.6+)

You might occasionally be required to let users download data from your webapp. If it's just a static file, this is a nonissue—just put in a link to the file on a web page. However, it is often the case that the data in question is produced on demand. In this case, you'll have to handle the downloading process yourself.

This isn't difficult (you need to write a valid HTTP header and then place the data into the `HttpServletResponse` object that comes with `execute()`), but it's a little tedious. The newly introduced (since 1.2.6) `DownloadAction` class simplifies this work. All you need to supply are

- An `InputStream` containing the data to be downloaded.
- A "content type" string (e.g., `text/html` or `image/png`) that describes the data's format. For a list of content type strings, take a look at the "Useful Links" section.

`DownloadAction` has an inner static interface called `StreamInfo`:

```
public static interface StreamInfo {
   public abstract String getContentType();
   public abstract InputStream getInputStream() throws IOException;
}
```

that exposes the content type via `getContentType()`, and the `InputStream` through `getInputStream()`. To use `DownloadAction`, you need to

- Implement `StreamInfo`.
- Expose it by subclassing `DownloadAction` and overriding its `getStreamInfo()` function.
- Put in a form handler (`<action>` tag) in the `struts-config.xml` file, associating your `DownloadAction` subclass with a path. No `<forward>` element is necessary. (Can you tell why?)

As a double bonus, `DownloadAction` comes with two inner classes, `FileStreamInfo` and `ResourceStreamInfo`, both of which implement the `StreamInfo` interface.

You can use `FileStreamInfo` to allow downloading of a static file. The constructor for `FileStreamInfo` is

```
public FileStreamInfo(String contentType, File file)
```

which is self-explanatory.

`ResourceStreamInfo` allows users to download a file within a webapp's root directory:

```
public ResourceStreamInfo(String contentType,
                          ServletContext context, String path)
```

The ServletContext contains information about a webapp's paths, among other things. You can obtain an instance for your webapp from the HttpSession object:

```
ServletContext context = response.getSession().getServletContext();
```

So, all you need to specify are the content type and the relative path to the file to be downloaded.

Of course, in most cases you might prefer to use HTML links instead of FileStreamInfo or ResourceStreamInfo. You'll find these classes useful if for some reason you can't expose a link to the file to be downloaded. Listing 17-3 shows an example of how you might subclass DownloadAction, implementing StreamInfo using an anonymous class.

Listing 17-3. *Extending DownloadAction*

```
package com.myco.myapp.struts;

import java.io.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import org.apache.struts.actions.DownloadAction;

public class MyDownloadAction extends DownloadAction{

   protected StreamInfo getStreamInfo(ActionMapping mapping,
                                      ActionForm form,
                                      HttpServletRequest request,
                                      HttpServletResponse response)
   throws Exception{

     /* Get InputStream somehow, as a result of processing request */
     InputStream input = ...;

     /* Similarly get content type */
     String contentType = ...;

     return new StreamInfo{
        public InputStream getInputStream(){ return input; }
        public String      getContentType(){ return contentType; }
     }

   }
}
```

You'll have to create a form handler as well:

```
<action path="/MyFileDownloadHandler"
        form="MyDataForm"
        type="com.myco.myapp.struts.MyDownloadAction" />
```

The form bean (`MyDataForm`) is only necessary if you need to collect user input in order to create the downloadable content. You don't need a `<forward>` because the downloaded content is the implicit "next" page.

Exactly what the user sees onscreen would depend on the browser. For example, suppose your download handler delivers a PDF file. If the user clicks the link invoking the handler, the browser might display the PDF file within the browser *if* there's an Adobe browser plug-in installed. If there isn't a plug-in installed, he might navigate to a blank page and get a download dialog box. The exact behavior would depend on the browser involved.

If the user clicks Save As instead of opening the file, then no navigation to a "next" page is performed. The user only gets a download dialog box. For this reason, you should mark such links with the type of data that's being downloaded so that the user can make the appropriate choice (whether to click the link or Save As).

LocaleAction

Toward the end of Chapter 12, I mentioned that there's another way to switch locales. The way to do it is to use the Struts class, `LocaleAction`. Unlike the server-side solution I outlined in Chapter 12, you don't have to write any Java code in order to use `LocaleAction`. Here's what you need to do:

- Implement dynamic forms, one for each locale you want to allow users to switch to. These forms *must* contain a `language` property representing the locale's language code (see Chapter 12). You must set the `initial` value of this property to the locale's language code (e.g., `jp` for Japanese, `en` for English). You may *optionally* specify a `country` attribute for the locale's country code (see Chapter 12). Again, the initial value of the `country` property must be set to the country code (e.g., `US`).

- Put in a form handler (`<action>` tag) for each of these dynamic forms. The single `<forward>` tag should be named `success` and must point to the page you want displayed in the new language.

As an example, suppose we want to allow users to switch between English or Japanese. The `struts-config.xml` file declarations are shown in Listing 17-4.

Listing 17-4. *Using LocaleAction*

```
<form-beans>

   <form-bean name="English"
              type="org.apache.struts.action.DynaActionForm">
      <form-property name="language" type="String" initial="en" />
      <form-property name="country"  type="String" initial="US" />
   </form-bean>

   <form-bean name="Japanese"
              type="org.apache.struts.action.DynaActionForm">
      <form-property name="language" type="String" initial="jp" />
   </form-bean>

   ... //other form beans

</form-beans>

<action-mappings>

   <action path="/ToEnglish"
           name="English"
           type="org.apache.struts.actions.LocaleAction">

       <forward name="success" path="/mystartpage.jsp" />
   </action>

   <action path="/ToJapanese"
           name="Japanese"
           type="org.apache.struts.actions.LocaleAction">

       <forward name="success" path="/mystartpage.jsp" />
   </action>

   ... //other form handlers

</action-mappings>
```

It goes without saying that `mystartpage.jsp` should be localized using Struts. The alternative is to use ordinary (but localized) HTML or JSP pages for the `<forward>`s.

To allow users to switch locales, you'd have to put in <html:link>s (see Appendix C) to the relevant form handler:

```
<html:link action="/ToJapanese">
   <bean:message key="prompt.to.japanese" />
</html:link>
<html:link action="/ToEnglish" >
   <bean:message key="prompt.to.english" />
</html:link>
```

LocaleAction makes the switch in locales for the user's session, using the same technique described in Chapter 12 (in the subsection "Switching Locales with a Link").

IncludeAction and ForwardAction

One compelling reason to use Struts is that it provides a framework for validating user input. This is good news if you have legacy code (business logic) in the form of servlets or JSPs: it's possible to include a Struts front-end to thoroughly validate user input before it reaches the legacy code.

You can easily roll your own front-end solution, but Struts has one you can use immediately. IncludeAction and ForwardAction make it easy to integrate Struts with legacy servlets or JSPs.

You'd use them like any other Action, with the addition of a parameter attribute in the <action> tag. This parameter attribute is a path pointing to the location of the servlet or JSP that contains the business logic you want to invoke, once Struts has finished validating user input.

For example, suppose your legacy code (a servlet) performs user login, and you want to put a Struts front-end to run simple validations, as in Chapter 6, Listing 6-1 (the Registration webapp). The <action> declaration using IncludeAction and linking LoginForm (to hold and validate user data) with a legacy servlet com.myco.myapp.MyLoginServlet would be

```
<action path="/Login"
        type="org.apache.struts.actions.IncludeAction"
        name="LoginForm"
        validate="true"
        input="/login.jsp"
        parameter="/WEB-INF/classes/com/myco/myapp/MyLoginServlet" />
```

Notice that just as with DownloadAction, you can't specify a <forward> because the legacy servlet is responsible for producing a "next" page. You *can*, however, specify exception handlers.

INTERNATIONALIZATION

Unfortunately, there's no easy way for you to internationalize legacy applications with this technique since you have little control over output of legacy code.

If your legacy code is well written, with business logic in servlets and view code in JSPs, then it's possible to migrate the legacy JSPs to using Struts tags and use `IncludeAction` or `ForwardAction` to mediate between Struts and the legacy servlets. This assumes, of course, that the legacy code correctly processes non-ASCII characters. This is often *not* the case, but if your target languages only use Latin-1, then you might be lucky enough that they do.

The `<action>` declaration using `ForwardAction` is similar—you just replace `org.apache.struts.actions.IncludeAction` with `org.apache.struts.actions.ForwardAction`.

This begs the question: what is the difference between the two?

`IncludeAction` performs a programmatic "include." This means two things:

- The called legacy code *cannot alter* cookies or header information sent to the client. The headers and cookies sent to the client are dictated by Struts. Preserving header or cookie information is crucial in some scenarios—for example, when you've specified the scope of a request with Struts.

- You may subclass `IncludeAction` to write content to the response object. Of course, you need to call `super.execute()` at some point to ensure that the legacy code is called. You can do this at any time, either before or after or in between writing content to the response object.

`ForwardAction` performs a programmatic "forward," meaning that *total control* over the data sent to the client is passed to the legacy code. Struts *cannot* dictate the header or cookie information, and you may *not* subclass `ForwardAction` in order to write data to the client, like you could with `IncludeAction`. `ForwardAction` simply passes all control of the response object to the legacy code.

In a Nutshell...

In most cases where you need to put a front-end to legacy code, you'd probably use `IncludeAction`, since this gives you greater flexibility. It's also useful since you can still write data to the client after the legacy code has sent its output.

`ForwardAction` is the only way out when the legacy code needs to write header or cookie information in order to work correctly (for example, if it needs to set the content type to something other than `text/html`).

LookupDispatchAction

You might sometimes want to put more than one submit button on a single form, each instructing the server-side code to perform a different action with the submitted data.

One common technique to accomplish this is to use JavaScript to set the value of a hidden field on the form, allowing the server-side processing code to tell which submit button was clicked by the user.

As an example, consider the page that follows, which features two submit buttons (Save and Update), a hidden field, and JavaScript to change the hidden field's value depending on which button was clicked:

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html>
<head>
  <script language="JavaScript">

    function changeMode(formName, src){
      document.forms[formName].elements["mode"].value = src;
    }

  </script>
</head>
<body>
  <html:form action="MyFormHandler.do">

    //... other fields on the form, omitted

    <html:hidden property="mode" value="unknown"/>

    <html:submit value="Save"
                onclick="changeMode('MyFormBean','save')" />

    <html:submit value="Update"
                onclick="changeMode('MyFormBean','update')" />
  </html:form>
</body>
</html:html>
```

This homebrew approach has three drawbacks:

- It won't work if the user has switched off JavaScript.
- Each time you had two or more submit buttons, you'd have to duplicate server-side code to dispatch processing according to the value of the `<html:hidden>` field. You could create your own `Action` subclass to do this, but as you'll see shortly, you'd be reinventing the wheel, and a poorer one at that.
- Another irritation is that you'd have to paste or "include" or "import" the JavaScript on each JSP that has this functionality. And you'd have to remember to put in the hidden field and the `onsubmit` properties for each submit button.

Struts gives you a way to avoid all this unpleasantness. `LookupDispatchAction` allows you to have multiple actions on your HTML form *without* having to code any JavaScript.

`LookupDispatchAction` does its magic by using the fact that when you use `<html:submit>` with a `property` attribute, the text displayed on the submit button is passed as a parameter in the URL when the form is submitted. The name of the parameter submitted is the value of the `property` attribute. This is how `LookupDispatchAction` knows which submit button was clicked on a form.

Using `LookupDispatchAction` is easy. We'll consider the concrete example of having a form with two submit buttons: Print to print the form data and Save to save the data.

The first thing you'll do is implement the form (see Listing 17-5).

Listing 17-5. *Constructing the Form*

```
<html:form action="MyFormHandler.do">

   ... // form properties here

   <html:submit property="action">
      <bean:message key="myapp.submit.button.print" />
   </html:submit>

   <html:submit property="action">
      <bean:message key="myapp.submit.button.save" />
   </html:submit>

</html:form>
```

Perhaps the only surprising thing about Listing 17-5 is the use of the `property` attribute in the `<html:submit>` buttons. These signal that Struts should pass the *text* of the submit button (in this case, the *value* of either `myapp.submit.button.print` or `myapp.submit.button.save`) as a parameter in the request URL, with the parameter name of `action`.

For example, if the Print button was clicked, the request URL would be

```
http://..../MyFormHandler.do?action=Print
```

assuming, of course that the value of `myapp.submit.button.print` in the user's currently selected locale is `Print`.

■ **Note** I've used `property="action"` in Listing 17-5, but you can use a different value.

`LookupDispatchAction` uses this information contained in the URL to tell which button was clicked. In real life, there may be other parameters passed in the URL, so you need to tell `LookupDispatchAction` *which* parameter name to use. This is done when you declare the form handler, as shown in Listing 17-6.

Listing 17-6. *Declaring the Form Handler*

```
<action path="/MyFormHandler"
        type="myco.myapp.struts.MyLookupDispatchAction"
        parameter="action"
        ...
```

Listing 17-6 shows how to declare the form handler. The only additional information is the `parameter="action"`, which tells your `LookupDispatchAction` subclass that it should use the parameter named `action` in the request URL to tell which submit button was clicked.

Lastly, you need to implement the `LookupDispatchAction` subclass, which processes the "Print" and "Save" requests. You also have to implement the function `getKeyMethodMap()`, which tells `LookupDispatchAction` which function to call when a particular submit button is clicked, as Listing 17-7 shows.

Listing 17-7. *Subclassing LookupDispatchAction*

```
import org.apache.struts.actions.LookupDispatchAction;
...
public class MyLookupDispatchAction extends LookupDispatchAction{

   /**
    * Tells LookupDispatchAction which function to
    * use when a given "submit" button is clicked.
    *
    * NOTE: This function is called only once by the
    * base class's execute(), so there's no need to
    * save a copy of the returned map.
    **/
```

```
    protected Map getKeyMethodMap(){
       Map m = new HashMap();
       m.put("myapp.submit.button.print","print");
       m.put("myapp.submit.button.save","save");
       return m;
    }

    public ActionForward print(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response)
    throws IOException, ServletException {

       //code to print here.
       //remember to return the "next" page.
    }

    public ActionForward save(ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response)
    throws IOException, ServletException {

       //code to save here.
       //remember to return the "next" page.
    }
}
```

As you can see in Listing 17-7:

- `getKeyMethodMap()` returns a `Map` instance, which contains the message keys used for each submit button's text. This message key is associated with the name of the function in your `LookupDispatchAction` subclass, which needs to be called. Note that `getKeyMethodMap()` is called only once, so there's no need to save a copy of the returned `Map` instance. In fact, the base class automatically saves the returned instance for you.

- There are functions, `print()` and `save()`, with the same signature as `execute()`. These do the work of printing and saving the submitted form data. Which function is called depends on the submit button clicked by the user.

- There is *no* implementation of `execute()`, since we want to implicitly use the base class's `execute()` in order to call either `print()` or `save()`.

That's all there is to using `LookupDispatchAction`, apart from a couple of loose strings:

- If the request URL contains no action parameter, then an `Exception` is thrown. Rather than leave this to chance, you might find it expedient to trap such errors by implementing a function called `unspecified()`, with the same signature as `execute()`. You should *not* declare this function in `getKeyMethodMap()`.

- Some forms use a cancel button. You can implement this using `<html:cancel>` on your form. You can handle a canceled form by implementing a function called `cancelled()`, with the same signature as `execute()`. You should *not* declare this function in `getKeyMethodMap()`.

Using `unspecified()` and `cancelled()`, Listing 17-7 could be rewritten as shown in Listing 17-8.

Listing 17-8. *MyLookupDispatchAction with unspecified() and cancelled()*

```
import org.apache.struts.actions.LookupDispatchAction;
...
public class MyLookupDispatchAction extends LookupDispatchAction{

   //no changes here
   protected Map getKeyMethodMap(){
      Map m = new HashMap();
      m.put("myapp.submit.button.print","print");
      m.put("myapp.submit.button.save","save");
      return m;
   }

   public ActionForward print(...)
   throws IOException, ServletException {

      //code to print here.

   }

   public ActionForward save(...)
   throws IOException, ServletException {

      //code to save here.

   }
```

```
    public ActionForward unspecified(ActionMapping mapping,
                                     ActionForm form,
                                    HttpServletRequest request,
                                    HttpServletResponse response)
    throws IOException, ServletException {

       mapping.findForward("page-not-found");
    }

    public ActionForward cancelled(ActionMapping mapping,
                                   ActionForm form,
                                   HttpServletRequest request,
                                   HttpServletResponse response)
    throws IOException, ServletException {
       mapping.findForward("main-page");
    }

}
```

`LookupDispatchAction` is an elegant alternative to using hidden fields for handling forms with multiple actions. It's also a more robust alternative since it does not depend on JavaScript being enabled on the client's machine.

DispatchAction

As the name implies, `DispatchAction` is the base class of `LookupDispatchAction`, and it performs method dispatching on a URL parameter.

The most common usage of `DispatchAction` is to call it from an HTML link, with the parameters embedded in the link. As with `LookupDispatchAction`, dispatching is done using the parameter name declared in the `<action>` tag using the `parameter` attribute:

```
<action path="/MyHandler"
        type="myco.myapp.struts.MyDispatchAction"
        parameter="command"
        ...
```

And again, for each possible value of the `command` parameter, there has to be a corresponding function. The difference here with `LookupDispatchAction` is that there is *no* `getKeyMethodMap()`. This is because the values of `command` in the URL must correspond exactly with functions defined in the `DispatchAction` subclass.

As a concrete example, suppose `command` can take the value `detailed` or `summary` (detailed or summarized versions of data for the user to view). The links might be

```
<a href="MyHandler.do?command=detailed&id=35 >View Details</a>
<a href="MyHandler.do?command=summary&id=35 >View Summary</a>
```

Your subclass of DispatchAction must implement the functions detailed() and summary(), with the same signature as execute(). You should *not* override execute(), and you may implement unspecified() and cancelled() if you wish, as Listing 17-9 shows.

Listing 17-9. *MyDispatchAction*

```
import org.apache.struts.actions.DispatchAction;
...
public class MyDispatchAction extends DispatchAction{

   public ActionForward detailed(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
   throws IOException, ServletException {

      //code for detailed view here.
   }

   public ActionForward summary(...)
   throws IOException, ServletException {

      //code for summarized view here.
   }

   public ActionForward unspecified(...)
   throws IOException, ServletException {

     mapping.findForward("page-not-found");
   }
}
```

MappingDispatchAction

This Action subclass helps you group related functionality in one Action. Like LookupDispatchAction, MappingDispatchAction is also a DispatchAction subclass. Like the former two classes, you also have to subclass the base class (MappingDispatchAction) in order to use it.

Unlike `LookupDispatchAction` or `DispatchAction`, `MappingDispatchAction` does not dispatch on a request parameter. Instead, you declare form handlers with the same `MappingDispatchAction` subclass to perform different actions.

As an example, suppose you wanted to group slightly different printing functionality in your webapp: Print to PDF, Print to HTML, and Print to Text. The `Action` subclass handling printing would be responsible for obtaining the necessary data and sending it off to appropriate helper classes in order to print to the right format. In this scenario, it would be unnatural to have the data be handled by three different `Actions`, since the code is likely to be very similar. You'd want to have *one* `PrintingAction` with three functions, one for each print format.

To do this, you declare three `<action>`s, one for each print format, as shown in Listing 17-10.

Listing 17-10. *Multiple Declarations for PrintingAction*

```
<action path="/PrintToPDF"
        type="myco.myapp.struts.PrintingAction"
        parameter="pdf"
        ...
<action path="/PrintToHTML"
        type="myco.myapp.struts.PrintingAction"
        parameter="html"
        ...
<action path="/PrintToText"
        type="myco.myapp.struts.PrintingAction"
        parameter="text"
        ...
```

The `parameter` attributes in Listing 17-10 point to actual function names on `PrintingAction` (see Listing 17-11).

Listing 17-11. *PrintingAction*

```
import org.apache.struts.actions.MappingDispatchAction;
...
public class PrintingAction extends MappingDispatchAction{

   public ActionForward pdf(ActionMapping mapping,
                            ActionForm form,
                            HttpServletRequest request,
                            HttpServletResponse response)
   throws IOException, ServletException {
```

```
        //code for printing to PDF here.
    }

    public ActionForward html(...)
    throws IOException, ServletException {

        //code for printing to HTML here.
    }

    public ActionForward text(...)
    throws IOException, ServletException {

        //code for printing to text here.
    }

    public ActionForward unspecified(...)
    throws IOException, ServletException {
        mapping.findForward("page-not-found");
    }
}
```

The signatures of `pdf()`, `html()`, `text()`, and `unspecified()` are the same for `Action`'s `execute()`.

■ **Note** As with other `DispatchActions`, you do *not* override `execute()`.

As you should know by now, `unspecified()` handles the case where the requested function does not exist on `PrintingAction`. (Question: how can this happen?)

In a Nutshell...

As you might have noticed, `MappingDispatchAction` and `DispatchAction` are very similar. Certainly, both dispatch according to the parameter on the URL. The difference is how you use them:

- `MappingDispatchAction`: Use this when you want to define multiple form handlers for the same `Action`.
- `DispatchAction`: Use this if for any reason you don't want to define multiple form handlers (e.g., to make `struts-config.xml` more manageable).

In most cases, choosing between them is largely a matter of taste. My preference is to use `MappingDispatchAction` throughout my apps because you can immediately tell what functionality is available on the associated `Action`, based on the declarations in `struts-config.xml`.

Using Global Forwards

In Chapter 9, I showed you how to declare global forwards. These forwards are accessible anywhere within the `struts-config.xml` file, *and* from `<html:link>`s. The latter is where global forwards come into their own.

Instead of hard-coding *paths* in your `<html:link>`s, you can use global forwards instead.

As an example, many webapps have common static navigational links. The LILLDEP main page is a good example. There are a few navigational links: Full, MNC, Listing, Import, and Collect. Instead of hard-coding these like

```
<a href="Listing.do">...
<a href="import.jsp">...
...
```

you should use

```
<html:link forward="listing">...
<html:link forward="import">...
...
```

and declare the global forwards as

```
<global-forwards>
   <forward name="listing" path="/Listing.do"/>
   <forward name="import" path="/import.jsp"/>
   ...
</global-forwards>
```

The advantage with this approach is that should you move a JSP or HTML page from one location to another, you don't have to amend every link to that page. You only need to change the global forward's path. This greatly improves the maintainability of your webapp.

Logging

Struts uses the Apache Commons Logging interface for logging. This is a unified *interface* to a number of logging systems.

When it's started, Commons Logging locates the best logging system for your installation (Log4j or the logging facilities of Java 1.4), and if neither is present, it uses the default `SimpleLog` class.

■ **Note** Log4j belongs to the Apache Logging subgroup. It is a very widely used open source logging system. See "Useful Links" for a URL.

You create logger instances per class—that's per class, *not* per instance of a class. For example, in order to log messages in a particular `Action` subclass, use the code in Listing 17-12.

Listing 17-12. *Logging with Apache Commons Logging*

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
...
public class MyAction extends Action{

    private static Log log = LogFactory.getLog(MyAction.class);

    public ActionForward execute(...){

        try{
           //some processing code

        }catch(IOException ioe){
           //log the exception
           if(log.isErrorEnabled()){
               log.error("IO Exception occurred!",ioe);
           }
        }
    }
}
```

`LogFactory` is a helper class used to create a `Log` instance, given the `Class` object associated with `MyAction`:

```
LogFactory.getLog(MyAction.class)
```

The returned value is a `Log` instance, which is a wrapper around the actual underlying logging class (Log4j or the JDK 1.4 logger or `SimpleLog`). It has a number of functions you can use to log different messages:

- `trace()`: Used to log very detailed information
- `debug()`: Used to log a debugging message
- `info()`: Used to signal "interesting" events at runtime, like startup or shutdown
- `warn()`: Used to show that something just short of an error has occurred
- `error()`: Used to flag an exception thrown during processing
- `fatal()`: Used to signal a fatal error, causing premature system shutdown

Each of these functions has two forms. The first (using `trace()` as an example)

```
trace(Object message)
```

prints out the given message, while the second form

```
trace(Object message, Throwable ex)
```

allows you to print an exception in addition to a message. The actual message finally logged depends on the underlying logging system used. Remember that Commons Logging is just a wrapper around the logging systems that do the real work.

Each of the functions listed here also represents a *priority level*: "trace" has lowest priority and "fatal" the highest. This priority system makes it possible to be selective about what messages to log from which class or package.

For example, Log4j allows you to define (using a properties file) exactly what priority messages are allowed to be logged by a given class or package, as shown in Listing 17-13.

Listing 17-13. *Some Settings for Log4j*

```
log4j.logger.com.myco.myapp=INFO
log4j.logger.com.myco.myapp.struts.database.Database=ERROR
```

The setting in Listing 17-13 means that the classes in the package `com.myco.myapp` are allowed to log fatal, error, warn, and info messages, but *not* debug or trace ones. The second line means that the `Database` class is only allowed to log fatal or error messages.

I'd like to stress that Listing 17-13 is merely illustrative, and does not apply to every logging system but just to Log4j. Each underlying logging system will have to be configured differently, and this *cannot* be controlled by Commons Logging. Commons Logging is only a *wrapper* around some properly configured logging system.

You should look up the documentation for the logging system you're using in order to determine how to configure it. Commons Logging only buys you portability. It doesn't make setting up the logging system simpler.

Now, because some classes may be configured with logging priorities below a given level, it makes sense to detect this first, before attempting a logging. This is mainly

because a call to a logging function (`trace()`, `debug()`, etc.) usually involves creating extra system resources. To avoid this unnecessary overhead, it's best to test first if logging at the desired level is allowed. The code:

```
if(log.isErrorEnabled()){
   log.error("IO Exception occurred!",ioe);
}
```

checks first that `MyAction` has logging clearance including and above `error`.

In a Nutshell...

Commons Logging is an easy-to-use, portable way for you to perform logging in your Struts (or other) applications. Use it rather than a given logging solution directly.

Using Wildcards

Wildcards are a way to cut down repetition in your `struts-config.xml`, as long as your webapp has some regular structure. For example, consider Listing 17-14, a variant of the declared `<action>`s corresponding to the Full and MNC pages for LILLDEP.

Listing 17-14. *A Variation on the LILLDEP Declarations for Full and MCN Page Handlers*

```
<action path="/ContactFormHandler_full"
        type="net.thinksquared.lilldep.struts.ContactAction"
        name="ContactFormBean"
        scope="request"
        validate="true"
        input="/full.jsp">

        <forward name="success" path="/full.jsp"/>
</action>

<action path="/ContactFormHandler_mnc"
        type="net.thinksquared.lilldep.struts.ContactAction"
        name="ContactFormBean"
        scope="request"
        validate="true"
        input="/mnc.jsp">

        <forward name="success" path="/mnc.jsp"/>
</action>
```

There's an obvious repeated structure in Listing 17-14. Using wildcards you can cut down the two declarations to just one, as shown in Listing 17-15.

Listing 17-15. *LILLDEP Declarations Take 2*

```
<action path="/ContactFormHandler_*"
        type="net.thinksquared.lilldep.struts.ContactAction"
        name="ContactFormBean"
        scope="request"
        validate="true"
        input="/{1}.jsp">

        <forward name="success" path="/{1}.jsp"/>
</action>
```

The * in the path is a wildcard that matches zero or more characters *excluding* the slash (/) character. You'd use ** to include the slash. You must make this distinction because the slash has a special meaning when used in paths (as you'll see in the next subsection).

In Listing 17-15, I've used just one wildcard. It's possible to use more than one; for example:

```
<action path="/*FormHandler_*" ...
```

would match paths like `ContactFormHandler_full` or `SearchFormHandler_mnc`. You'd access the matched strings using `{1}` and `{2}`. With `ContactFormHandler_full`, `{1}` would equal `Contact` and `{2}` would equal `full`. In all, you are allowed nine wildcards.

■ **Note** The `{0}` wildcard returns the full request URI.

If more than one `<action>` matches a request, then the last one declared in `struts-config.xml` is used. There's an exception to this rule: if an `<action>` contains no wildcards and it matches a request, it is *always* used.

In a Nutshell...

When you name your `<action>`s, take the effort to name them with an eye to future wildcard use. Don't go overboard with wildcards, though, because they make your system less manageable.

A good rule to follow is to use at most one wildcard per `<action>`.

Splitting up struts-config.xml

As your project grows, and especially if you work in a team, you might find yourself wishing that it were possible to break up `struts- config.xml` into several pieces. There are a few reasons why you might want to do this:

- **Manageability**: As the `config.xml` file becomes too big, the control flow of your webapp becomes less transparent. You'd want to break up `struts-config.xml` into separate sections, each of which perhaps represents control flow of a limited portion of the overall webapp.

- **Separate namespaces**: You have two or more teams working on different aspects of the webapp, and you'd like these teams to work independently of each other. The primary concern here is to give each team different *namespaces*. Distinct namespaces would help avoid the problem of two teams creating two different form beans with the same name.

- **Source control**: Some poorly designed source control systems allow only single checkouts. If you're working in a large team, this is potentially very restrictive, since it's likely that more than one person might want to edit `struts-config.xml` at the same time.

The first way to split up `struts-config.xml` is to simply do just that. You can create multiple `struts-config.xml` files (with different names, of course!) and instruct Struts that they are to be regarded as one file. To do this, you have to edit `web.xml`, the servlet configuration file.

For example, suppose you had multiple configuration files: `struts-config.xml`, `struts-config-admin.xml`, and `struts-config-logon.xml`. Listing 17-16 shows how you'd declare them both in `web.xml`. All you need to do is to put a comma between each Struts configuration file. Easy!

Listing 17-16. *Declaring Multiple Struts Configuration Files in web.xml*

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
     org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
      <param-name>config</param-name>
```

```
      <param-value>
        /WEB-INF/struts-config.xml,
        /WEB-INF/struts-config-admin.xml,
        /WEB-INF/struts-config-logon.xml
      </param-value>
  </init-param>
</servlet>
```

Now, this approach solves manageability and source control issues, but it does not solve namespace problems because Struts will internally merge together all the declared configuration files. So, although they appear as two or more files to you, they appear as one to Struts.

There is a Struts feature called modules that allows you to split a Struts configuration files into different modules. A module represents a *logical* split in your webapp. Each module gets its own namespace, so this solves namespace issues. Using the previous example, suppose we want to give each Struts config file separate namespaces: the default namespace for items declared in `struts-config.xml`, the `admin` namespace for `struts-config-admin.xml`, and the `logon` namespace for `struts-config-logon.xml`. The resulting declaration in `web.xml` appears in Listing 17-17.

Listing 17-17. *Declaring Multiple Submodules in web.xml*

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
     org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
      <!-- THE DEFAULT SUB-MODULE -->
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
      <!-- THE "ADMIN" SUB-MODULE -->
      <param-name>config/admin</param-name>
      <param-value>/WEB-INF/struts-config-admin.xml</param-value>
  </init-param>
  <init-param>
      <!-- THE "LOGIN" SUB-MODULE -->
      <param-name>config/logon</param-name>
      <param-value>/WEB-INF/struts-config-logon.xml</param-value>
  </init-param>
</servlet>
```

With the declaration in Listing 17-17, we may declare two different form beans with the same name in two different submodules. The same applies to <action>s, <forward>s, and other configuration elements.

Access to <action>s and <forward>s would differ between submodules. For example, suppose we had an <action> declared in the admin submodule:

```
<action path="Listing.do" ...
```

Then, you'd refer to it as /admin/Listing.do in your <html:link>s or <html:form>s.

■ **Note** If you had the same <action> defined in the default submodule, then you'd use /Listing.do.

This brings up an important issue: in any well-designed application, you're likely to have shared declarations like shared global <forward>s or form beans. When you split your app using multiple submodules, these are *not* shared between modules. To overcome this problem, you should group together shared declarations in one Struts configuration file.

For example, continuing our earlier example, let's call this new file struts-config-shared.xml. Listing 17-18 shows what you would do in order to share the declarations in this file across submodules.

Listing 17-18. *Using a Shared Struts Configuration File Across Submodules*

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
     org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
      <!-- THE DEFAULT SUB-MODULE -->
      <param-name>config</param-name>
      <param-value>
          /WEB-INF/struts-config-shared.xml,
          /WEB-INF/struts-config.xml
      </param-value>
  </init-param>
  <init-param>
      <!-- THE "ADMIN" SUB-MODULE -->
      <param-name>config/admin</param-name>
```

```
        <param-value>
            /WEB-INF/struts-config-shared.xml,
            /WEB-INF/struts-config-admin.xml
        </param-value>
    </init-param>
    <init-param>
        <!-- THE "LOGIN" SUB-MODULE -->
        <param-name>config/logon</param-name>
        <param-value>/WEB-INF/struts-config-logon.xml</param-value>
    </init-param>
</servlet>
```

In Listing 17-18, I've shared `struts-config-shared.xml` across the default and `admin` subapplications. This uses the same trick as the multiple configurations file technique described earlier. You just have to separate the filenames with a comma.

In a Nutshell...

There are two ways to split up your Struts configuration file. Use the multiple files approach if your concerns are only about manageability or source control. Use the multiple submodules approach if you need separate namespaces as well.

In the latter, you can declare common form beans, actions, forwards, and so forth in a single file and share them between submodules. This prevents duplication of declarations while giving you separate namespaces.

Useful Links

- A compendium of commonly used content type (MIME types) strings: `http://www.utoronto.ca/webdocs/HTMLdocs/Book/Book-3ed/appb/mimetype.html`
- Apache Commons Logging: `http://jakarta.apache.org/commons/logging/`
- Log4j: `http://logging.apache.org/log4j/docs/`

Summary

Struts provides a number of useful classes, features, and tricks to make writing webapps easier. This chapter covers some of the ways you can take advantage of what's available rather than reinventing the wheel in your webapps.

CHAPTER 18

Review Lab: The Collection Facility

In this lab session, you'll implement a facility to group together contacts into logical **collections**. This is essentially a tool to classify and manage contacts within LILLDEP.

There are four parts in this Collection facility:

- The main **Collect** page displays a list of defined collections as links. When a link is clicked, the user is given a listing of `Contacts` within that collection. The Collect page also allows new collections to be defined. Users access this page through the Collect navigation button (see Figure 18-1).
- The **New Collection** page prompts the user for a collection name, query, and memo. This page results in a listing of the newly defined collection's `Contacts`.
- The **Collection Listing** page lists the `Contacts` within a collection. It also allows the user to add an arbitrary `Contact` from a full listing or click the company name to get into the Collection Full Display page.
- The **Collection Full Display** page displays the `Contact`'s details, with Previous and Next buttons to navigate up and down the collection.

There are *two* database tables used to store data for a single collection. The `collection` table stores the collection's name, memo, and autogenerated collection ID. The `collection_map` table holds the `Contact` ID and collection ID as pairs.

■ **Note** We'll be using Lisptorq-generated Model classes to store and retrieve data from these tables, so be sure to read Appendix A's section on Lisptorq.

There's quite a bit to implement, so we've broken down this lab into six subparts. The source code answers for this lab (found in the Source Code section of the Apress website, located at `http://www.apress.com`) are similarly divided into six parts.

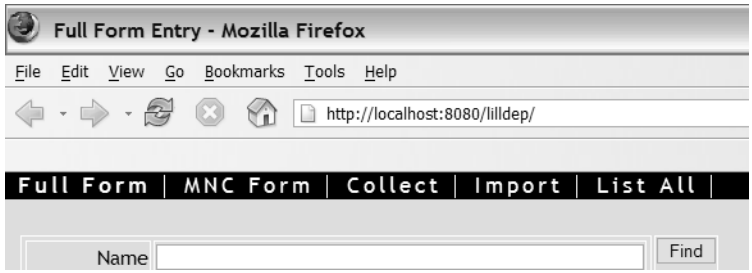
Full Form Entry - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/lilldep/
Full Form | MNC Form | Collect | Import | List All |
Name
Find

Figure 18-1. *The navigation button for the Collection facility*

Lab 18a: The Main Collect Page

The main Collect page allows users to create, view, and edit collections. Figure 18-2 shows this page with a couple of collections defined. This lab sets up the basic page, and subsequent labs implement the viewing and editing facilities.

1. Complete the implementation of `CollectAction` to prepare the list of collections to be displayed. (*Hint*: You need to use the `Criteria` class to retrieve all `Collection` data objects stored in the database.) Use an appropriate constant from `JSPConstants` for an attribute name under which to store the list of `Collection` data objects.

2. Put in an action mapping so that a link to `Collect.do` displays the list of collections. The `success` forward should point to `collect.jsp`.

3. Complete `collect.jsp` so that it displays the names of all defined collections. (*Hint*: The `Scroller` interface extends `Iterator`, so you should be able to use `<logic:iterate>` with it.)

4. Each collection listed by `collect.jsp` should have a delete link on its left linked to `DeleteCollection.do`. This link should pass the collection's ID as a parameter called `id`.

5. Complete the implementation of `DeleteCollectionAction` so that it deletes the selected collection. (*Hint*: Refer to Appendix B to see how to retrieve the value of the `id` parameter from `HttpServletRequest`.)

6. Put in an action mapping so that `DeleteCollection.do` maps to `DeleteCollectionAction`. `success` should forward to `Collect.do` (and *not* `collect.jsp`).

Ensure that your work compiles before proceeding.

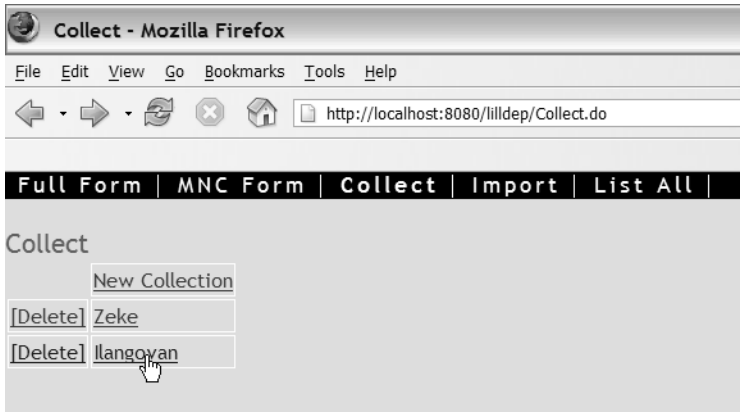
Collect - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/lilldep/Collect.do
Full Form | MNC Form | Collect | Import | List All |
Collect
New Collection
[Delete] Zeke
[Delete] Ilangovan

Figure 18-2. *The main Collect page, showing some collections*

Lab 18b: The New Collection Page

Each collection has a *name* by which it is known and a *memo* for holding the user's notes. In addition, a collection contains any number of `Contacts`. In order to bootstrap a collection, the user needs to specify a SQL *query*:

- The queries are SQL-like—for example, `postcode<>'' and email like'%mit%'` would fetch all contacts with nonblank postcodes and whose email addresses contained the string `mit`.

- Each query results in a list of contacts matching the query string. The user may subsequently add, remove, or edit contacts within the collection.

Figure 18-3 shows the New Collection page.

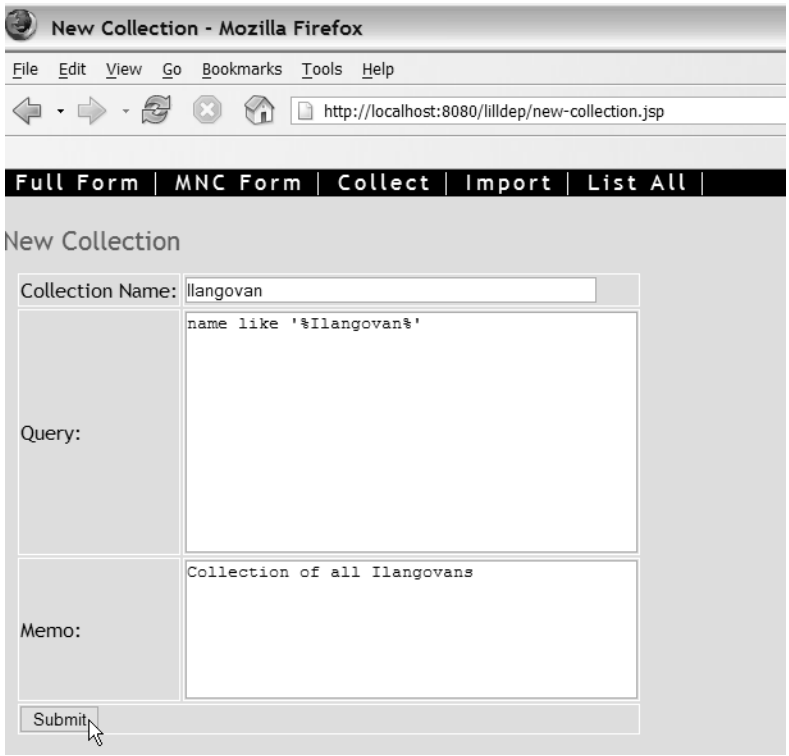

Figure 18-3. *The New Collection page*

Complete the following:

1. Put in a new **dynamic** form bean called `NewCollectionFormBean`, with the properties `name`, `query`, and `memo` as `String`s.

2. Put in validators (using the Validator framework) to check that the `name` and `query` attributes are not blank. Check the message resource file for suitable error messages.

3. Complete the implementation of `new-collection.jsp` so that it asks the user for the `name`, `query`, and `memo` property values. Remember to put in `<html:error>` tags so that the user is warned if a validation fails.

4. Complete the implementation of `NewCollectionAction` to create a new collection. (*Hint*: Use `DynaActionForm`'s generic `get()` to read property values. You also need to know how to run an arbitrary SQL `SELECT` using a peer object.)

5. Put in an action mapping to create a new collection. `success` forwards to `list-collection.jsp`.

6. Declare a *local* exception handler on the action mapping to pass to `new-collection.jsp` if there's an error. This is exactly the same as declaring a global exception handler (see Chapter 9), except that the `<exception>` tag appears as a child tag of `<action>`. Note that the `<exception>` tag has to come before any `<forward>` tags.

Lab 18c: The Collection Listing Page

You will now implement functionality to view the contacts in a collection. Figure 18-4 shows the Collection Listing page, after you've completed Lab 18d. First, you need to enable listing the collection from the main Collect page:

1. On `collect.jsp`, make the collection's name link to `ListCollection.do`, with the collection's ID as a parameter.

2. Put in an action mapping for `ListCollection.do`, linking it with `ListCollectionAction`. `success` forwards to `list-collection.jsp`.

3. `list-collection.jsp` lists the contacts in a collection, but it needs a `Scroller` to iterate through all the contacts. Complete the implementation of `ListCollectionAction` so that the selected `Collection` is put on the session (see `JSPConstants` for a suitable key), thus making it accessible to `list-collection.jsp`. The reason for putting it on the session rather than the request will become obvious in Labs 18d and 18f. Compile your work before proceeding.

4. `NewCollectionAction` also forwards to `list-collection.jsp`, so it also needs to put the newly created `Collection` on the session. Use the same key as in step 3.

5. Complete `list-collection.jsp` so that it displays the list of `Contacts` on a `Collection`. You should list exactly the same items as the Full Listing page (see Chapter 13). Display the name of the collection prominently at the top of the page. (*Hint*: You need to use `Collection.getContacts()`.)

Compile, deploy, and test before proceeding. In particular, test that you can

- Create a new collection
- List a newly created collection
- Access the collection listing from the main Collect page
- Delete a collection from the main Collect page

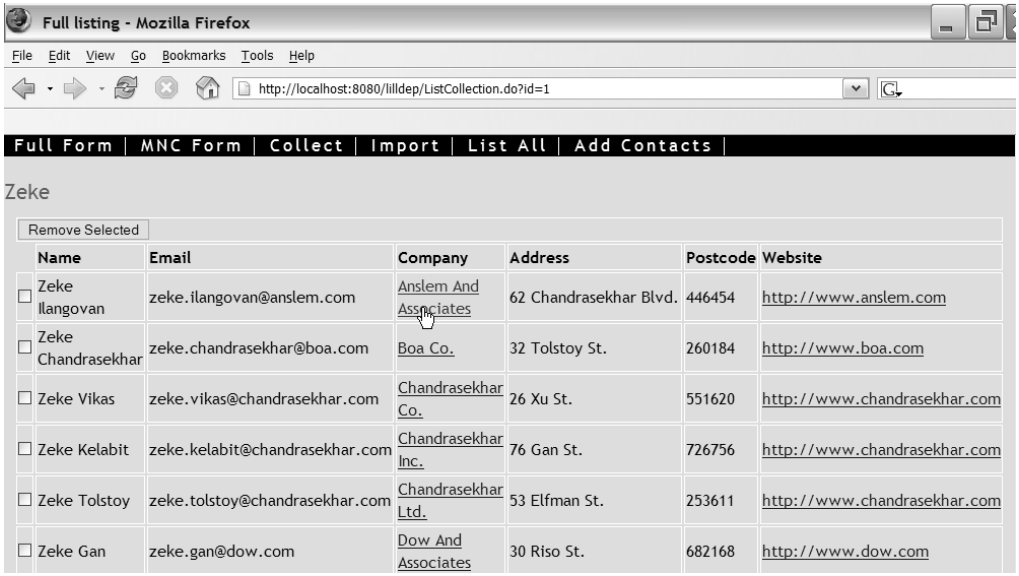
Full listing - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/lilldep/ListCollection.do?id=1
Full Form | MNC Form | Collect | Import | List All | Add Contacts |
Zeke
Remove Selected
Name Email Company Address Postcode Website
Zeke Ilangovan zeke.ilangovan@anslem.com Anslem And Associates 62 Chandrasekhar Blvd. 446454 http://www.anslem.com
Zeke Chandrasekhar zeke.chandrasekhar@boa.com Boa Co. 32 Tolstoy St. 260184 http://www.boa.com
Zeke Vikas zeke.vikas@chandrasekhar.com Chandrasekhar Co. 26 Xu St. 551620 http://www.chandrasekhar.com
Zeke Kelabit zeke.kelabit@chandrasekhar.com Chandrasekhar Inc. 76 Gan St. 726756 http://www.chandrasekhar.com
Zeke Tolstoy zeke.tolstoy@chandrasekhar.com Chandrasekhar Ltd. 53 Elfman St. 253611 http://www.chandrasekhar.com
Zeke Gan zeke.gan@dow.com Dow And Associates 30 Riso St. 682168 http://www.dow.com

Figure 18-4. *The Collection Listing page*

Lab 18d: Removing Selected Contacts

We want to add check boxes to the left of each contact's name (see Figure 18-4) in the collection listing (that is, `list-collection.jsp`), to allow the user to selectively remove contacts from a collection. In Lab 16 we used a dynamic form with `<html:multibox>`. We'll use the same technique now.

One approach is to create a new form (perhaps a dynamic form, following Lab 16), which holds the contacts to be removed from the collection, as well as the ID of the collection on which to perform this action:

```
<form-bean name="RemoveSelectionFormBean"
           type="org.apache.struts.action.DynaActionForm">

    <form-property name="selected"     type="java.lang.String[]" />
    <form-property name="collectionId" type="java.lang.String" />


</form-bean>
```

The latter might be stored as a hidden input field on the listing displayed to the user, while the `selected` property is populated using the `<html:multibox>`, as before.

This simple technique has an obvious drawback if you're worried about security, since a malicious user could manually craft an HTML form that had a collection ID to which he had no access privileges.

One simple way to prevent this is to store the collection ID on the session object instead of exposing it on the form. This way, a malicious user would have to fake an active session ID in order to gain access to a given collection. This is much more difficult than reading the collection ID embedded in the page's HTML.

■ **Note** One bonus is that we can reuse `SelectionFormBean` from Lab 16.

We'll use this approach here. In fact, we've had this technique in mind already—remember that in Lab 18c, we asked you to store the `Collection` instance in the session scope? Well, this is where that comes in handy. We can just use this object to determine the currently selected `Collection`.

The drawback is that we have to ensure that *all* entry points to `list-collection.jsp` must place the `Collection` onto the current session; otherwise, `list-collection.jsp` will not display correctly. There are two such entry points—`NewCollectionAction` and `ListCollectionAction`—and both have done this already in Lab 18c.

So, all we have to do is the actual work of removing contacts from a collection:

1. Complete `RemoveCollectionContactsAction` to remove selected contacts from the given collection. You'll need to interrogate the current session object for the `Collection` you saved in the previous lab.

2. Put in an action mapping for a path `RemoveCollectionContacts.do` to link `RemoveCollectionContactsAction` and `SelectionForm` (see Lab 16). Forward to `list-collection.jsp`, so that the collection's listing is redisplayed.

Lastly, you need to amend `list-collection.jsp` so that it allows the user to select and submit contacts for removal:

1. Put in an `<html:multibox>` whose values are the IDs of the associated `Contact`.

2. Create an `<html:form>` that submits to `RemoveCollectionContacts.do`.

As usual, compile, deploy, and test your changes.

Lab 18e: Adding Selected Contacts

We need to allow the user to add selected contacts to a collection, while she is viewing the collection's listing (that is, `list-collection.jsp`). She does this by clicking Add Contacts on the navbar at the top of the page (see Figure 18-4).

We want the Add Contacts link to display a list of all contacts (displayed by `select-contacts.jsp`), with check boxes on the left of the company names. The user selects the contacts she wants to add to the current collection and this information gets sent to `AddCollectionContacts.do`. Figure 18-5 shows the Add Contacts page.

1. Create a new form handler called `AddContacts` that invokes `ListingAction` and forwards to `select-contacts.jsp`.

2. Complete `select-contacts.jsp` so that it displays a list of `Contacts` (as in the Full Listing page). The user is allowed to select any number of `Contacts` from this list to add to the currently selected `Collection`. Use the `<html:multibox>` technique you used in Lab 18d. Ensure that your form submits to `AddCollectionContacts.do`.

3. Complete the implementation for `AddCollectionContactsAction` so that it adds the selected `Contacts` to the current `Collection`.

4. Put in an appropriate form handler to hook up `AddCollectionContactsAction` with `AddCollectionContacts.do`. The form bean associated with `AddCollectionContacts` is, of course, `SelectionFormBean` (see Lab 16). `AddCollectionContacts` should forward to `list-collection.jsp`.

As usual, compile, deploy, and test your work before proceeding.

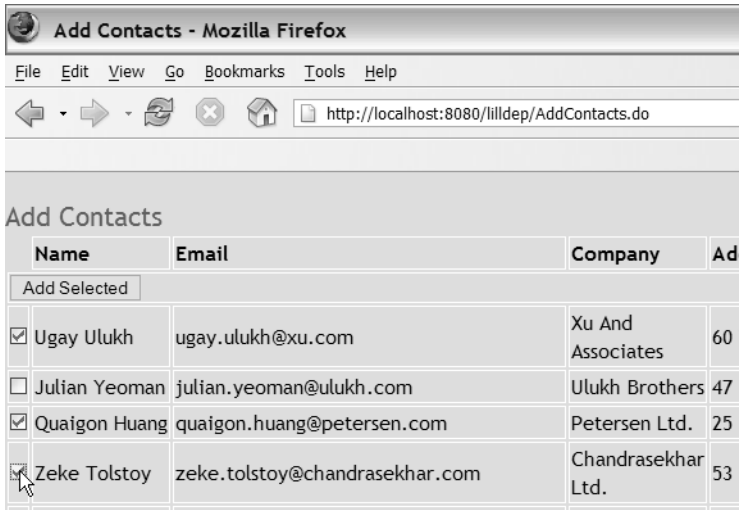

Figure 18-5. *The Add Contacts page*

Lab 18f: Up and Down a Search

From a collection's listing page (`list-search.jsp`), we'd like to edit the contacts in a collection, and navigate with Previous and Next buttons.

That is, we want the collection's listings to be a list of links (as it was for the Full Listing page in Chapter 13—see also Figure 18-4). If a link was clicked, the contact's details would be displayed. The user can move up and down the collection, viewing the details of each contact using Next or Previous buttons at the top of the page. Figure 18-6 shows the Contact Editing page. The user can at any point make changes to a contact's details. Doing so would cause the changes to be saved to the database, and the *same* contact's details to be redisplayed.

The main design challenge is accessing a given contact on the collection (when the user clicks the link), then navigating forward/backward along the collection to display other contacts for editing (when the user clicks Next or Previous).

Before reading further, give yourself a few minutes to think of solutions to these problems. (*Hint*: You will have to look at the source for the `Scroller` interface.)

The first problem is easily solved, since the `Scroller` interface has an `absolute()` function that allows us to go to the required contact on a collection, given the contact's "offset" on the collection.

■ **Note** `Scroller`'s `absolute()` function is so named because concrete `Scroller` implementations contain a `java.sql.ResultSet` instance, which also has an `absolute()` function. `Scroller`'s `absolute()` function calls the underlying `ResultSet`'s `absolute()` function.

Also, `<logic:iterate>` exposes an index variable (using the attribute `indexId`; see Chapter 10), which we can use to label the links on the listing. The `absolute()` function, along with an offset may be used to navigate up and down the collection's listing. As we will see, this is simpler than it sounds!

We'll use `DispatchAction` (see Chapter 17) here, so be sure you understand how it works before proceeding. Complete the following:

1. Amend `list-collection.jsp` so that the company name is now a link, pointing to `CollectionNav.do&action=go`, with a parameter called `offset` equal to the iterator's index, and a parameter called `id` for the contact's ID.

2. Complete the implementation of `CollectionNavAction` so that if `action=go`, it initializes the form (a `ContactFormBean`) with the `Contact` determined from the given `id` parameter. Save the `offset` parameter on the session under a suitable key.

3. Put in an action mapping for the path `CollectionNav.do`, linking `CollectionNavAction` and the form bean `ContactFormBean`. The forward should be to `full-collection.jsp`. Remember to set the `parameter` attribute of the action mapping.

4. Complete the implementation of `ContactUpdateAction` to save the updated contact.

5. Put in an action mapping to handle submission of the form in `full-collection.jsp`. Reuse `ContactForm` to hold the contact details. Remember to use session scope so that the `Contact` will be redisplayed if Submit is clicked.

6. Amend `CollectionNavAction` to handle `action=previous` and `action=next`. These obviously are commands to go up and down the `Collection` contact list. (*Hint*: Use the `offset` saved in `CollectionNavAction`'s `go()` function. You can use this `offset` and the `absolute()` function to navigate to the right contact.)

Compile, deploy, and test your amendments.

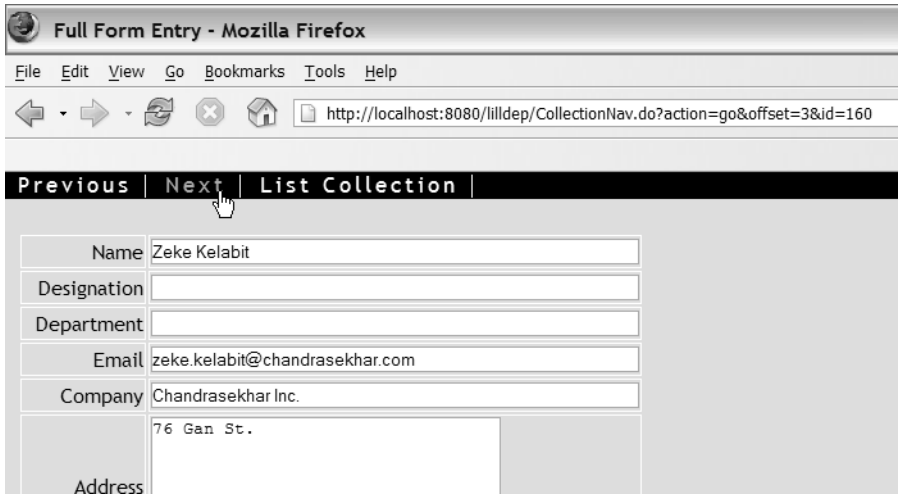

Figure 18-6. *The Contact Editing page*

Summary

I hope this review lab reinforces some of the concepts you've learned in the second part of this book. I also hope you see how easy it is to incrementally build on an existing webapp using Struts.

CHAPTER 19

Developing Plug-ins

Plug-ins are a great way to extend the basic functionality of Struts. In previous chapters, you've seen plug-ins at work—in Tiles and the Validator framework. In this chapter, I'll walk you through developing a nontrivial and useful plug-in called DynaForms. Essentially, the DynaForms plug-in brings Tiles-like inheritance to dynamic form beans. I'll explain in more detail what this means shortly.

The inspiration for this idea comes from an excellent pair of articles: "Adding Spice to Struts," parts 1 and 2, by Samudra Gupta (see "Useful Links" at the end of this chapter). Samudra's solution to the problem of implementing an inheritance mechanism for dynamic form beans involves subclassing fundamental Struts classes, primarily `ActionServlet`. Unfortunately, as Samudra discusses in his article, his solution has a couple of downsides: the inheritance declaration is a little kludgy and limited. If you're interested in learning more, look up these articles for details.

In this chapter, we'll take a different route and, I hope, a more scenic one—meaning you get to see more of how Struts works under the hood. We'll create a plug-in that allows developers to create form beans with inheritance in XML files *outside* of `struts-config.xml`.

In order for you to better follow the discussion, it would be helpful to have a copy of the latest Struts sources handy (see "Useful Links" for the download site; I'll be using release 1.2.7 in this chapter), and have them in a project within Eclipse (or a similar IDE). An IDE like Eclipse (again, see "Useful Links") is absolutely essential to easily trace function calls, declarations, and class hierarchies. You can do it with Notepad, but it won't be much fun.

The Task at Hand

Consider the hierarchy of entities shown in Figure 19-1 (a variation on the main example in Samudra's first article).

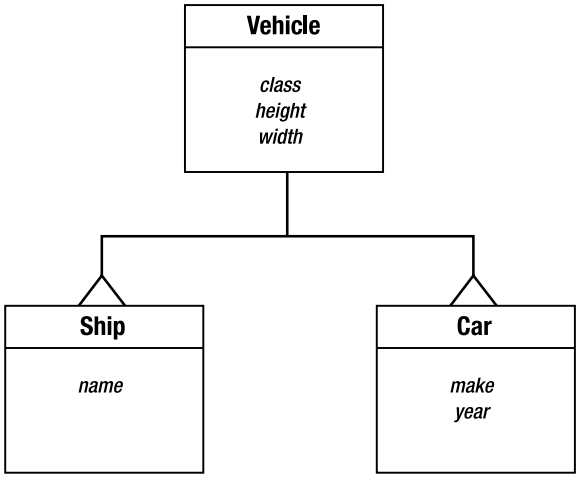

Figure 19-1. *A simple hierarchy of entities*

Both the `Ship` and `Car` entities contain properties from `Vehicle`. Now, suppose you wanted to create dynamic form beans corresponding to each of these entities. How would you do it? Listing 19-1 shows how.

Listing 19-1. *Dynamic Form Bean Declaration for Entity Hierarchy*

```
<form-bean name="Vehicle"
           type="org.apache.struts.action.DynaActionForm">

    <form-property name="class" type="java.lang.String"
                   initial="Unknown Class"/>
    <form-property name="height" type="java.lang.Integer"
                   initial="0"/>
    <form-property name="width"  type="java.lang.Integer"
                   initial="0"/>
</form-bean>

<form-bean name="Ship"
           type="org.apache.struts.action.DynaActionForm">
    <form-property name="class"  type="java.lang.String"
                   initial="Ship"/>
    <form-property name="height" type="java.lang.Integer"
                   initial="1000"/>
    <form-property name="width"  type="java.lang.Integer"
                   initial="0"/>
    <form-property name="name"   type="java.lang.String"/>
</form-bean>
```

```
<form-bean name="Car"
           type="org.apache.struts.action.DynaActionForm">
    <form-property name="class"  type="java.lang.String"
                   initial="Car"/>
    <form-property name="height" type="java.lang.Integer"
                   initial="1000"/>
    <form-property name="width"  type="java.lang.Integer"
                   initial="0"/>
    <form-property name="make"   type="java.lang.String"/>
    <form-property name="year"   type="java.lang.Integer"/>
</form-bean>
```

There's a lot of duplication in the declarations. This becomes much worse if there are more "subentities" of `Vehicle`. Our goal is to allow Listing 19-1 to be simplified as shown in Listing 19-2.

Listing 19-2. *Using the extends Attribute*

```
<form-bean name=".vehicle"
           type="org.apache.struts.action.DynaActionForm">
   <form-property name="class"  type="java.lang.String"
                  initial="Unknown Class"/>
   <form-property name="height" type="java.lang.Integer" initial="0"/>
   <form-property name="width"  type="java.lang.Integer" initial="0"/>
</form-bean>

<form-bean name=".vehicle.ship" extends=".vehicle">
   <form-property name="class"  initial="Ship"/>
   <form-property name="height" initial="1000"/>
   <form-property name="name"   type="java.lang.String"/>
</form-bean>

<form-bean name=".vehicle.car"  extends=".vehicle">
   <form-property name="class"  initial="Car"/>
   <form-property name="make"   type="java.lang.String"/>
   <form-property name="year"   type="java.lang.Integer"/>
</form-bean>
```

Much better! As you can see, this really cuts down the code duplication. The `extends` attribute functions like its counterpart in Tiles, where you could "extend" one Tiles definition from another. Notice that I've used a dot naming system for the form names, similar to the one used for Tiles.

We'll want to have a few other things in the DynaForms plug-in:

- A "sub form bean" can override both the `initial` value or the `type` of a `<form-property>`.
- DynaForms must place no restrictions on the class type of the form bean. You should be able to declare *any* form bean with DynaForms.
- There should be a mechanism to prevent "ancestor" form beans, which are never used in your `struts-config.xml` file, from being instantiated. This saves system resources. To accomplish this, I'll add another attribute called `create` into `<form-bean>`. Setting `create=false` would suppress creation of that form bean. For example, if we wanted to suppress creation of `.vehicle` in Listing 19-2, we'd use `<form-bean name=".vehicle" create="false" ....`

In summary, apart from the `extends` and `create` attributes, the new form bean declarations look just like the original Struts ones.

In addition, I'll make one big simplification: the new form bean declarations won't support the `<set-property>` tag. Adding support for `<set-property>` is not *too* difficult. You are encouraged to attempt this on your own at the end of the chapter.

Lastly, we can't place our new form beans in `struts-config.xml` itself, since this will require subclassing `ActionServlet`. Instead, we'll have to place them in another XML file (or in multiple XML files).

Now that the goals of the DynaForms plug-in are clear, the next step is deciding how to achieve them.

Implementation Road Map

Essentially, the implementation of the DynaForms plug-in would have to

- Parse the input XML files (there might be more than one). So the plug-in declaration will require at least a `pathnames` property, like the Validator framework, which is a comma-separated list of filenames.
- Resolve the inherited properties of a form bean.
- Somehow get Struts to recognize these as if they had been declared in `struts-config.xml`.

For the first, I'll show you how to use the Apache Digester, which is an easy way to read XML files. Digester is also used internally by Struts. The second step is relatively straightforward, but the last requires some knowledge of how Struts reads and declares form beans. We'll tackle this last step first.

How Struts Processes Form Beans

When Struts first loads, a single instance of `ActionServlet` is created, which handles all subsequent initialization of Struts, including reading of `struts-config.xml`. This class is internal to Struts, but the servlet container knows about it since it is declared in `web.xml`. Listing 19-3 shows the declaration in `web.xml`. You can also see that the name of the Struts configuration file, `struts-config.xml`, is passed as an initialization parameter into `ActionServlet`.

■ **Note** In case it isn't clear by now, Listings 19-3, 19-4 and 19-5 are all excerpts from code generously made available by the Apache Software Foundation. The Apache License is available at `http://www.apache.org/licenses/LICENSE-2.0`.

Listing 19-3. *Declaring the ActionServlet and struts-config.xml file in web.xml*

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
     org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
     <param-name>config</param-name>
     <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
</servlet>
```

Once the `ActionServlet` instance is created, its `init()` function is called to initialize Struts with the information contained in `struts-config.xml`. The relevant code that actually reads `struts-config.xml` appears in Listing 19-4.

Listing 19-4. *Initializing Struts in init()*

```
public void init() throws ServletException {

   ...

   // Initialize modules as needed
   ModuleConfig moduleConfig = initModuleConfig("", config);
   initModuleMessageResources(moduleConfig);
   initModuleDataSources(moduleConfig);
   initModulePlugIns(moduleConfig);
   moduleConfig.freeze();

   ...
}
```

In Listing 19-4, you should note a few things:

- `config` refers to the filename of the Struts configuration file (`/WEB-INF/struts-config.xml` by default).
- `ModuleConfig` is a class that holds all configuration information contained in `struts-config.xml`. It has several variables, each of which holds information from a different part of `struts-config.xml`. For example, form bean declarations are held in a variable of `ModuleConfig` of type `FormBeanConfig`. Figure 19-2 shows how form bean data is stored in `ModuleConfig` through the use of `FormBeanConfig` and `FormPropertyConfig`.
- Form beans are read in at the function `initModuleConfig()`.
- Plug-ins are processed in sequence by `initModulePlugIns()`. As you will see shortly, all plug-ins will get a chance to interrogate and manipulate the `ModuleConfig` instance.
- The `ModuleConfig` instance is "frozen" after the `struts-config.xml` file is read. No changes may be made to a `ModuleConfig` instance after it has been frozen.

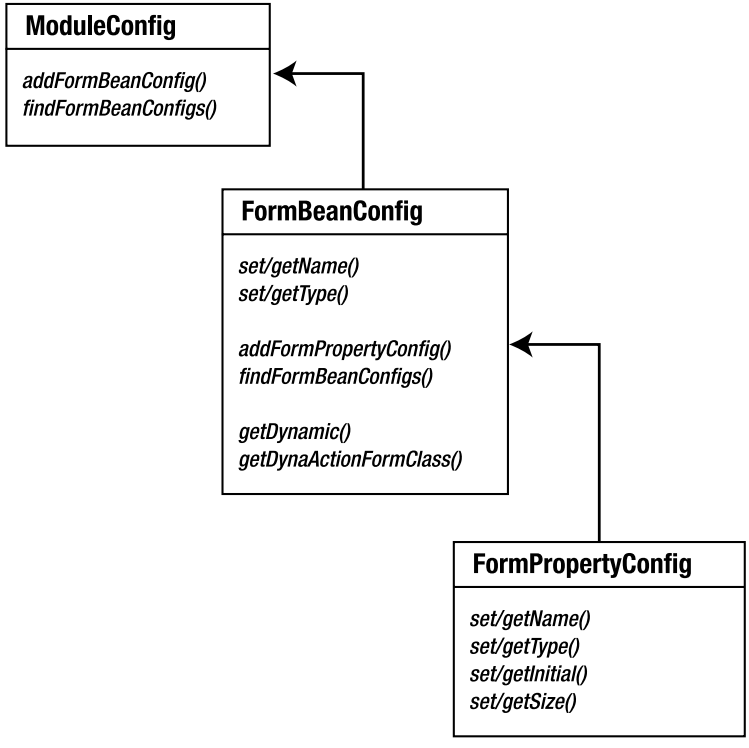

Figure 19-2. *ModuleConfig, FormBeanConfig, and FormPropertyConfig*

A ModuleConfig instance is passed to our plug-in as soon as it is created. Our DynaForms plug-in will have to create new FormBeanConfig instances based on the new XML form bean declaration files and add them to ModuleConfig. Once this is accomplished, we're done.

Well, almost! A quick check on initModuleConfig(), which creates a ModuleConfig instance and initializes it with form bean information, shows that form beans whose type are subclasses of DynaActionForm require special handling (see Listing 19-5).

Listing 19-5. *Special Treatment for DynaActionForm Subclasses*

```
protected ModuleConfig initModuleConfig(String prefix, String paths)
throws ServletException {

   /* create ModuleConfig instance, "config" */

   /* read form-bean data using Apache Digester */

   // Force creation and registration of DynaActionFormClass instances
   // for all dynamic form beans we will be using
   FormBeanConfig fbs[] = config.findFormBeanConfigs();
   for (int i = 0; i < fbs.length; i++) {
       if (fbs[i].getDynamic()) {
           fbs[i].getDynaActionFormClass();
       }
   }

   return config;

}
```

In Listing 19-5, config refers to the newly created ModuleConfig instance. As the listing shows, once this instance has been filled with form bean information (held by FormBeanConfig instances), each FormBeanConfig instance is interrogated (using getDynamic()) to see if it is a subclass of DynaActionForm. If so, the function getDynaActionFormClass() is called on that FormBeanConfig. All getDynaActionFormClass() does is initialize that DynaActionForm subclass with the <form-property>s stored in FormBeanConfig.

We will have to duplicate this behavior in our DynaForms plug-in.

EXERCISE

Open `ActionServlet` in Eclipse (or some other IDE) and follow the function calls I've described in this section. Also look at `ModuleConfig` and `FormBeanConfig`.

Your goal should be to satisfy yourself of the veracity of the statements I've made in this section.

Now that you have a global view of how form beans are read and stored, I'll show you how to create a plug-in.

Anatomy of a Plug-in

A plug-in must implement Struts' `PlugIn` interface, which is shown in Listing 19-6.

Listing 19-6. *The PlugIn Interface*

```
/*********************************************************************
Licensed under the Apache License, Version 2.0 (the "License"); you may not use
this file except in compliance with the License. You may obtain a copy of the
License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed
under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and limitations
under the License.
*********************************************************************/

package org.apache.struts.action;

import javax.servlet.ServletException;
import org.apache.struts.config.ModuleConfig;

public interface PlugIn {

    //called after the PlugIn has been initialized using
    //setters for each property
```

```
    void init(ActionServlet servlet, ModuleConfig config)
        throws ServletException;

    //called just before system shutdown
    void destroy();

}
```

This interface has just two functions:

- `init()` is called when the `PlugIn` instance is created. Note that this function receives an instance of `ModuleConfig`.
- `destroy()` is called when Struts shuts down, to allow the plug-in to clean up any system resources (release file handles, close database connections, etc.).

In addition to these two, your implementation of the `PlugIn` interface will have additional setters for each parameter that you want to pass into your plug-in class. These parameters are set in `struts-config.xml` in the plug-in's declaration. For example, the declaration

```
<plug-in className="com.mycompany.myapp.MyPlugIn" >
   <set-property property="message" value="hello world"/>
</plug-in>
```

implies that `MyPlugIn` has a `setMessage()` function.

In the following section, I'll introduce you to `DynaFormsPlugIn`.

Implementing DynaFormsPlugIn

Listing 19-7 is an implementation of the `PlugIn` interface for the DynaForms plug-in.

Listing 19-7. *DynaFormsPlugIn.java*

```
package net.thinksquared.struts.dynaforms;

import java.io.IOException;
import java.util.StringTokenizer;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
```

```
import org.apache.struts.action.PlugIn;
import org.apache.struts.action.ActionServlet;
import org.apache.struts.config.ModuleConfig;


public class DynaFormsPlugIn implements PlugIn{

    protected String _pathnames = "";

    public void setPathnames(String pathnames){
        _pathnames = pathnames;
    }

    public void init(ActionServlet servlet, ModuleConfig config)
    throws ServletException{

        ServletContext context = servlet.getServletContext();

        StringTokenizer tokenizer =
             new StringTokenizer(_pathnames,",");

        while(tokenizer.hasMoreTokens()){

            String fname =
               context.getRealPath(tokenizer.nextToken().trim());

            try{

                DynaFormsLoader loader =
                    DynaFormsLoaderFactory.getLoader(fname);

                loader.load(fname,config);

            }catch(NoSuitableLoaderException le){
                throw new ServletException(le);
            }catch(IOException ioe){
                throw new ServletException(ioe);
            }catch(DefinitionsException de){
                throw new ServletException(de);
            }
        }
```

```
    }

    public void destroy(){/* do nothing */}

}
```

As you can see, `DynaFormsPlugIn` contains three functions: a setter for the `pathnames` property, which is a comma-delimited list of filenames (see the section "Implementation Road Map" earlier). These filenames refer to files that contain the form bean declarations. The `destroy()` function has a "no-op" implementation (meaning it does nothing).

Most of the action is in the `init()` function. This function does the following:

- Gets a reference to `ServletContext`. This class is needed to resolve file paths. In our case we have XML files containing form bean declarations, which we have to read. The plug-in-declaration will only specify them relative to the webapp's root directory, for example, `/WEB-INF/form-beans-def.xml`. Obviously, this path isn't enough. To read the file, we need the full path to the file. This is where `ServletContext` comes in. It has a `getRealPath()` function that resolves the full path of a given relative file path.

- Gets a `DynaFormsLoader` instance from the `DynaFormsLoaderFactory`. `DynaFormsLoader` is actually an interface that we need to implement. This is good design: if you needed to load form beans with your *own* `DynaFormsLoader`, you can. The `DynaFormsLoaderFactory` determines which loader to use from the `<form-beans>`'s `type` attribute. You'll see how this is done shortly.

- The `load()` function on `DynaFormsLoader` does the actual work of reading the form bean declarations and adding the necessary `FormBeanConfig` instances to `ModuleConfig`.

To summarize, the `init()` function parses the `pathnames` variable, using a comma delimiter. Each file's full path is determined using the `ServletContext`'s `getRealPath()` function. This filename is passed to `DynaFormsLoaderFactory`, which gives us an instance of `DynaActionForm`. We call `load()` on this instance to parse and load the form bean declarations into `ModuleConfig`. The `load()` function will also resolve the `extends` properties of each form bean, and run the special processing of `DynaActionForms`, described in the earlier section "How Struts Processes Form Beans."

Before we can proceed, you have to know how to read XML files with Apache's Digester framework. This easy-to-use framework is utilized extensively within Struts. The JAR files for the Digester framework are in the Struts distribution (`commons-digester.jar`). This is another good reason to use Digester rather than another XML parsing tool (like the excellent JDOM).

Reading XML with Apache's Digester

Apache's Digester framework gives you a simple way to read portions of an XML document. The data that is read in is stored in JavaBeans that you supply to Digester.

Usually, you have to define one JavaBean class for each type of tag being read, but this isn't a requirement. You *can* have one JavaBean for more than one tag. But in this chapter, we'll use one JavaBean per tag.

You also have to instruct Digester *which* portions of the XML file you want to read. These portions are stored in your JavaBeans objects. For example, suppose your XML file, which is called `myfile.xml`, contains *just* the tag shown here:

```
<user id="albert" password="relativity" desc="Albert Einstein" />
```

Now you want to read this data into the `User` JavaBean shown in Listing 19-8.

Listing 19-8. *The User Bean*

```
public class User{

   protected String _id = null;
   protected String _password = null;

   public void setId(String id){ _id = id; }
   public String getId(){ return _id; }

   public void setPassword(String pwd){ _password = pwd; }
   public String getPassword(){ return _password; }
}
```

Notice that the `desc` attribute isn't stored. Listing 19-9 shows how to do it with Digester.

Listing 19-9. *Using Digester to Create a Single User*

```
//Step 1: Create new Digester instance:
   Digester digester = new Digester();

//Step 2: Associate tag <user> with User bean:
   digester.addObjectCreate("user", "User");

//Step 3: Tell Digester to read in all
//        required properties into User bean:
   digester.addSetProperties("user");
```

```
//Step 4: Actually parse the file
   User u = (User) digester.parse(new File("myfile.xml"));

//Step 5: Our User bean contains the data from the XML file!
   System.out.println("ID=" + u.getId()  + " pwd=" + u.getPassword());
```

The `parse()` function on the Digester class (`org.apache.commons.digester.Digester`) parses the input XML file and returns a populated `User` instance. There are two functions you should take special note of:

- `addObjectCreate(path, bean)`: Takes a path in the XML file and a JavaBean that will get populated based on that path. In our earlier example, the path is just `user`, which we want to read from and return a populated `User` object.

- `addSetProperties(path)`: This tells Digester that you want to read the attributes of the tag indicated by the given path into the JavaBean you associated with that path using `addObjectCreate()`. So, in our previous example, if you omitted `addSetProperties()`, the returned `User` bean would contain no information.

Handling more than one `user` declaration is also easy. For example, suppose now we amended our `myfile.xml` as shown in Listing 19-10.

Listing 19-10. *The Amended XML File, with Multiple Users*

```
<users>
  <user id="albert" password="relativity" desc="Albert Einstein" />
  <user id="marie" password="radioactivity" desc="Marie Curie" />
  <user id="paul" password="qed" desc="Paul Dirac" />
</users>
```

To read in all the `Users` on this file, we could store them into a `Users` class, as Listing 19-11 shows.

Listing 19-11. *The Users Bean*

```
public class Users extends java.util.ArrayList{

   public void addUser(User u){
      add(u);
   }

   public User getUser(int i){
      return (User) get(i);
   }
}
```

As you can see in Listing 19-11, the `Users` bean is just a collection of `User` beans. We want Digester to create `User` beans based on the new XML (see Listing 19-8) and place them into the containing `Users` bean. To do this, we'll have to amend Listing 19-9 as shown in Listing 19-12.

Listing 19-12. *Using Digester to Create Multiple Users*

```
//Step 1: Create new Digester instance:
   Digester digester = new Digester();

//Step 2: Associate <users> tag with Users class.
   digester.addObjectCreate("users", "Users");

//Step 3: Associate <user> tag with User bean.
//        Note that the path to the <user> tag is
//        indicated.
   digester.addObjectCreate("users/user", "User");

//Step 4: Tell Digester to read in all
//        required properties into User bean:
   digester.addSetProperties("users/user");

//Step 5: Tell Digester to put all newly created
//        User beans into the Users class by calling
//        Users.addUser(aNewUserbean)
   digester.addSetNext("users/user", "addUser");


//Step 6: Actually parse the file
   Users users = (Users) digester.parse(new File("myfile.xml"));

//Step 7: Print out each User in the Users bean:
   for(int i = users.size() - 1; i >= 0; i--){
     User u = users.getUser(i);
     System.out.println("ID=" + u.getId()  +
                        " pwd=" + u.getPassword());
   }
```

Apart from the obvious changes to the paths and the bean returned by `parse()`, the most significant change is calling Digester's `addSetNext(path, myFunction)` function.

When Digester finishes parsing a single `<user>` tag, it asks "What's next?" This answer is given by the information you supplied in `addSetNext()`. Digester will call `myFunction()` on the object associated with the *parent* of the path defined by `addSetNext()`. The argument

of `myFunction()` is the latest object created. This is the key to understanding how `addSetNext()` works, and it can be a little confusing, so let's take it one step at a time.

Looking again at Listing 19-12, the instruction

```
digester.addObjectCreate("users", "Users");
```

maps the path `/users` (I've put in the leading / to indicate it's a path) to the class `Users`. So, each time Digester meets a `/users` tag, it creates a new `Users` object. Let's jump forward:

```
digester.addSetNext("users/user", "addUser");
```

This tells Digester that after it has parsed the tag with the path `/users/user` (and therefore created a `User` bean), it has to call the `addUser()` function on the object associated with the *parent path* of `/users/user`. The parent path of `/users/user` is obviously `/users`, and we know the object associated with the `/users` path is the `Users` object. So, the function `addUser()` is called on this `Users` bean, with the latest `User` as the argument of `addUser()`.

Note that there's no need to implement a loop in order to read each `User`. Each time the Digester meets a path matching one you declared using `addObjectCreate()`, `addSetProperties()`, or `addSetNext()`, it takes the appropriate action indicated by the function.

This section only skims the surface of the Digester framework. If you're interested in learning more, consult the many articles on the Internet describing this useful framework in more detail.

We next take a look at the implementation of `DynaFormsLoaderFactory`, the class that creates a concrete implementation of `DynaFormsLoader`.

Implementing DynaFormsLoaderFactory

Listing 19-13 shows how `DynaFormsLoaderFactory` is implemented.

Listing 19-13. *DynaFormsLoaderFactory.java*

```
package net.thinksquared.struts.dynaforms;

import java.io.IOException;
import java.io.File;
import org.apache.commons.digester.Digester;
import net.thinksquared.struts.dynaforms.definitions.Beans;

public class DynaFormsLoaderFactory{

  private DynaFormsLoaderFactory(){}
```

```
public static DynaFormsLoader getLoader(String filename)
throws IOException,DefinitionsException,NoSuitableLoaderException{

  Beans config = readConfig(filename);

  String clazzname = config.getType();

  if( null == clazzname ){

    return new DefaultDynaFormsLoader();

  }else{
    try{
      return (DynaFormsLoader) instantiate(clazzname);
    }catch(Exception e){
      throw new NoSuitableLoaderException("Can't initialize loader "
                                          + clazzname);
    }
  }
}

protected static Beans readConfig(String filename)
throws IOException,DefinitionsException{

  try{

    Digester digester = new Digester();
    digester.addObjectCreate("form-beans",
             "net.thinksquared.struts.dynaforms.definitions.Beans");

    digester.addSetProperties("form-beans");

    return (Beans)digester.parse(new File(filename));

  }catch(IOException ioe){
    throw ioe;
  }catch(Exception e){
    throw new DefinitionsException("Definitions file " +
                                    filename +
                                   " has incorrect format");
  }
}
```

```
  protected static Object instantiate(String clazzname)
  throws Exception{
    return Class.forName(clazzname).newInstance();
  }
}
```

First, notice that the constructor has only `private` access. This is a design pattern called *Singleton* and it prevents `DynaFormsLoaderFactory` from being instantiated. Instantiating `DynaFormsLoaderFactory` is a waste of resources, since all the action goes on in *static* functions on this class.

THE SINGLETON DESIGN PATTERN

In its original formulation, the Singleton design pattern was a solution to the problem of ensuring that *only one* instance of a class was made. But the essence of the Singleton design pattern is that it allows you to control the number of instances of a given class, hence my use of this term in the text.

To do this, all constructors of the class are marked as `private`, so the class cannot be directly instantiated using the `new` keyword. Instead, the class provides a static function (usually called `getInstance()`) that controls instantiation of the class. The following code gives a skeleton of a typical singleton:

```
public class MySingleton{
   private static MySingleton _self = new MySingleton();

   private MySingleton(){
      /* initialization code here */
   }

   public static MySingleton getInstance(){
      return _self;
   }
}
```

Notice that I've had to create the `_self` instance at the start. One other option is to test for a `_self == null` in `getInstance()` and only then create the `MySingleton` instance. This approach might fail in a multithreaded application, unless `getInstance()` is marked as `synchronized`.

The only `public` function is `getLoader()`, which returns a concrete implementation of `DynaFormsLoader`. This function takes as a parameter the filename of the form beans declaration file. With

```
Beans config = readConfig(filename);
```

the input file is parsed using the Digester framework. The `Beans` class is a JavaBean class, which we will come to later.

A look at the `readConfig()` function shows that only the root `<form-beans>` is read, in order to determine the `type` attribute. If the `type` attribute is not present, the factory returns an instance of `DefaultDynaFormsLoader` to load the form beans. You can specify your own implementation of `DynaFormsLoader` by putting in a value for `type`:

```
<form-beans type="com.mycompany.myapp.MyDynaFormsLoader"> ...
```

This flexibility can come in handy when you want to add new functionality to the DynaForms plug-in, like handling `<set-property>` tags.

In the next section, we'll delve into the details of the default implementation of `DynaFormsLoader`.

DefaultDynaFormsLoader

`DefaultDynaFormsLoader` (see Listing 19-14) reads the XML file using the Digester framework, parsing the data into a `Beans` class (see Listing 19-15), which holds all the form bean declarations from a single file.

Listing 19-16 describes the `Bean` class that holds information from a single `<form-bean>` declaration. It also has a function called `merge()` that creates properties on a "sub form bean," which have been declared in the parent form bean. Listing 19-17 describes the `Property` class, which holds the information from a single `<form-property>` declaration.

Listing 19-14. *DefaultDynaFormsLoader.java*

```
package net.thinksquared.struts.dynaforms;

import java.io.IOException;
import java.io.File;
import java.util.Iterator;

import org.apache.struts.config.ModuleConfig;
import org.apache.struts.config.FormBeanConfig;
import org.apache.struts.config.FormPropertyConfig;
import org.apache.commons.digester.Digester;

import net.thinksquared.struts.dynaforms.*;
import net.thinksquared.struts.dynaforms.definitions.*;

public class DefaultDynaFormsLoader implements DynaFormsLoader{
```

```
public void load(String filename,ModuleConfig config)
throws IOException, DefinitionsException{

  Beans beans = readBeans(filename);

  for(Iterator beanz = beans.values().iterator(); beanz.hasNext();){
    Bean b = resolve(beans, (Bean)beanz.next());
    if(b.getCreate()){
      config.addFormBeanConfig(createFormBeanConfig(b));
    }
  }

}

protected Beans readBeans(String filename)
throws IOException, DefinitionsException{
  try{

    Digester digester = new Digester();

    digester.addObjectCreate("form-beans",
             "net.thinksquared.struts.dynaforms.definitions.Beans");

    digester.addSetProperties("form-beans");

    digester.addObjectCreate("form-beans/form-bean",
              "net.thinksquared.struts.dynaforms.definitions.Bean");

    digester.addSetProperties("form-beans/form-bean");

    digester.addObjectCreate("form-beans/form-bean/form-property",
          "net.thinksquared.struts.dynaforms.definitions.Property");

    digester.addSetProperties("form-beans/form-bean/form-property");

    digester.addSetNext("form-beans/form-bean/form-property",
                        "addProperty");

    digester.addSetNext("form-beans/form-bean","addBean");

    return (Beans)digester.parse(new File(filename));
```

```
    }catch(IOException ioe){
      throw ioe;
    }catch(Exception e){
      throw new DefinitionsException("Definitions file " +
                                      filename +
                                     " has incorrect format.");
    }
  }

  protected Bean resolve(Beans beans, Bean b)
  throws DefinitionsException{

    if(b.getExtends() == null){

      //check that all properties have types
      Object[] properties = b.values().toArray();
      for(int i = 0; i < properties.length; i++){
        Property p = (Property) properties[i];
        if(p.getType() == null){
          throw new DefinitionsException("Type of property '" +
                                          p.getName() +
                                "' has not been set on form bean '" +
                                          b.getName() +
                                         "'");
        }
      }
      return b;
    }

    Bean parent = beans.getBean(b.getExtends());
    if(parent == null){
      throw new DefinitionsException(
                  "Can't resolve parent bean named " +
                   b.getExtends());
    }

    b.merge(parent);

    //resolve this bean further up the hierarchy
    return resolve(beans,b);

  }
```

```
  protected FormBeanConfig createFormBeanConfig(Bean bean){

    FormBeanConfig config = bean.getFormBeanConfig();

    //add in all the form-properties for this form-bean
    for(Iterator ps = bean.values().iterator(); ps.hasNext();){
      Property p = (Property) ps.next();
      config.addFormPropertyConfig(p.getFormPropertyConfig());
    }

    //Force creation and registration of DynaActionFormClass
    //instances for all dynamic form beans
    if(config.getDynamic()){
      config.getDynaActionFormClass();
    }

    return config;
  }
}
```

Listing 19-15. *Beans.java*

```
package net.thinksquared.struts.dynaforms.definitions;

import java.util.*;

public class Beans extends HashMap{

    protected String _type = null; //Loader classname

    public void addBean(Bean b){
        put(b.getName(),b);
    }

    public Bean getBean(String name){
        Object obj = get(name);
        return (obj == null)? null : (Bean)obj;
    }

    public void setType(String t){
        _type = t;
    }
```

```
    public String getType(){
        return _type;
    }

}
```

Listing 19-16. *Bean.java*

```
package net.thinksquared.struts.dynaforms.definitions;

import java.util.*;
import org.apache.struts.config.FormBeanConfig;

public class Bean extends HashMap{

    protected String _name = null;
    protected String _type = null;
    protected String _extends = null;
    protected boolean _create = true;


    public void addProperty(Property p){
        put(p.getName(),p);
    }

    public Property getProperty(String name){
        Object obj = get(name);
        return (obj == null)? null : (Property)obj;
    }

    public void setName(String n){
        _name = n;
    }

    public String getName(){
        return _name;
    }

    public void setType(String t){
        _type = t;
    }
```

```
public String getType(){
    return _type;
}

public void setExtends(String e){
    _extends = e;
}

public String getExtends(){
    return _extends;
}

public void setCreate(boolean b){
    _create = b;
}

public boolean getCreate(){
    return _create;
}


/**
* Merges a bean with its parent.
* if the bean b is not a parent, then
* the merging is NOT done.
*/
public void merge(Bean b){

    //don't merge if there is no common properties
    //between the beans.
    if(_extends == null || !_extends.equals(b.getName())) return;

    _extends = b.getExtends();

    if(_type == null){
        _type = b.getType();
    }

    Object[] properties = b.values().toArray();
```

```
        for(int i = 0; i < properties.length; i++){
            Property p = (Property) properties[i];
            Object obj = getProperty(p.getName());
            if(obj == null){
                addProperty(p);
            }else{
                Property cp = (Property) obj;
                if(cp.getType() == null){
                    cp.setType(p.getType());
                }
            }
        }
    }

    /**
    * Gets the FormBeanConfig instance
    * that this Bean represents
    */
    public FormBeanConfig getFormBeanConfig(){

        FormBeanConfig config = new FormBeanConfig();
        config.setName(getName());
        config.setType(getType());

        return config;
    }
}
```

Listing 19-17. *Property.java*

```
package net.thinksquared.struts.dynaforms.definitions;

import org.apache.struts.config.FormPropertyConfig;

public class Property{

    protected String _name = null;
    protected String _type = null;
    protected String _initial = null;
    protected int    _size = 0;
```

```
    public void setName(String n){
        _name = n;
    }

    public String getName(){
        return _name;
    }

    public void setType(String t){
        _type = t;
    }

    public String getType(){
        return _type;
    }

    public void setInitial(String i){
        _initial = i;
    }

    public String getInitial(){
        return _initial;
    }

    public void setSize(int s){
        _size = s;
    }

    public int getSize(){
        return _size;
    }

    /**
    * Gets the FormPropertyConfig instance
    * that this Property represents
    */
    public FormPropertyConfig getFormPropertyConfig(){

        return new FormPropertyConfig(getName(),
                                      getType(),
                                      getInitial(),
                                      getSize());
    }
}
```

Essentially, DefaultDynaFormsLoader's getLoader() first reads all the form beans from the given file:

```
Beans beans = readBeans(filename);
```

The beans object is a collection of Bean objects (see Listing 19-16), which in turn may have Property objects (see Listing 19-17). The beans object corresponds to the <form-beans> root tag. The Bean class corresponds to a <form-bean> declaration, and the Property class corresponds to a <form-property> tag within <form-bean>.

At this point, each bean object in the bean's container is *unresolved*, meaning that the bean's inherited properties have yet to be determined. The next bit of code does just this:

```
for(Iterator beanz = beans.values().iterator(); beanz.hasNext();){
  Bean b = resolve(beans, (Bean)beanz.next());
  if(b.getCreate()){
    config.addFormBeanConfig(createFormBeanConfig(b));
  }
}
```

This section of code iterates through all bean objects in the bean's container, resolving each one, then creating a FormBeanConfig instance from this resolved bean. Lastly, this FormBeanConfig object is added to the ModuleConfig instance using addFormBeanConfig(). Note that

- The resolve() function recursively merges the properties of a bean with that of an ancestor bean. The actual merging of properties is performed in bean.merge() (see Listing 19-16).
- createFormBeanConfig() creates a FormBeanConfig object from a given resolved bean.
- Form beans that are marked create=false (bean.getCreate() returns false) are *not* added to ModuleConfig.

The source code for the DynaForms plug-in is in the Source Code section of the Apress website at http://www.apress.com, in the file DynaFormsPlugIn.zip, and released under the General Public License (GPL).

Lab 19: Test Driving the DynaForms Plug-in

In this lab session, you'll take the DynaForms plug-in for a test drive.

1. Copy the file dynaformstest.zip from the Source code section of the Apress website into your development directory and unzip it.
2. Besides the usual Struts JAR files in the lib folder, you'll see the dynaforms.jar file, containing the DynaForms plug-in classes.

3. Take a look at the `struts-config.xml` file. Can you tell what file contains the new form bean declarations?

4. The `dynaformstest` webapp models the hierarchy of Listing 19-2. Compile and deploy the `dynaformstest` webapp. Play with the webapp and ensure that you understand how it works. Consult the source code for the DynaForms plug-in in `DynaFormsPlugIn.zip` if necessary.

Extra Credit Lab: Handling <set-property>

In this lab, you'll extend the DynaForms plug-in so that it handles `<set-property>` tags in form bean declarations. This tag is used to pass initialization parameters into custom `FormBeanConfigs` or `FormPropertyConfigs`.

According to the Struts configuration file 1.2 DTD (available in your Struts distribution), there are two places `<set-property>` occurs within a form bean declaration:

- As a child node of `<form-bean>`
- As a child node of `<form-property>`

In both cases, Struts assumes that there's a `setXXX` function corresponding to each `<set-property>`, on the custom `FormBeanConfig` or `FormPropertyConfig`. This is similar to how Struts expects a `setXXX` function on the `PlugIn` implementation class for each `<set-property>` tag used in a plug-in declaration.

Of course, such `setXXX` functions won't exist on either `FormBeanConfig` or `FormPropertyConfig`; they are assumed to exist on custom *subclasses* of these two.

By default, Struts will use the base `config` objects (`FormBeanConfig` or `FormPropertyConfig`) to hold form bean and form property configuration information. However, if you specify the `className` attribute in either tag, then the class referred to by `className` is used instead. For example, the information from

```
<form-bean name="ordinaryFormBean" ...
```

which *doesn't* specify a `className` attribute, will be placed in the default `FormBeanConfig`. But the information from

```
<form-bean name="specialFormBean"
           className="com.myco.myapp.MyFormBeanConfig" ...
```

will be placed in the class `MyFormBeanConfig`, which *must* be a subclass of `FormBeanConfig`.

So, this declaration

```
<form-bean name="specialFormBean"
           className="com.myco.myapp.MyFormBeanConfig">

   <set-property property="message" value="Hello World"/>
   ...
```

will cause Struts to instantiate `MyFormBeanConfig`, then call `setMessage("Hello World")` on this object.

Struts uses the Digester framework to call the appropriate `setXXX` functions on the `MyFormBeanConfig` object. Unfortunately, since we use our own custom `Bean` object to store the form bean data, we cannot take this approach.Instead, I'll outline a possible solution, which I encourage you to implement.

Solution Outline

1. Add `get/setClassName()` functions to both the `Bean` and `Property` classes. This allows Digester to store the `className` information.

2. Add a `HashMap` called `_properties` to both the `Bean` and `Property` classes. This will store the property/value pairs from a `<set-property>`.

3. Create a new class called `Parameter` in the DynaForms `definitions` directory. This class corresponds to a `<set-property>` tag, and should have getters and setters for `property` and `value`.

4. Add identical `addParameter(Parameter)` functions to both the `Bean` and `Property` classes. This function should take the property/value data in a `Parameter` and put the data on the `_properties` `HashMap`.

5. Obviously, we want the `<set-property>`s to be inherited, too, just like `<form-property>`s. To do this, amend the `merge()` function on the `Bean` class so that it copies a property/value pair from the parent bean into the child bean, *only if* the property doesn't exist on the child bean.

6. We need to get Digester to call the `addParameter()` functions of step 4 when it encounters a `<set-property>` tag. Amend `DefaultDynaFormsLoader`'s `readBeans()` function so that it calls Digester's `addObjectCreate()` to create a `Parameter` object. We need to call this twice, for a `<set-property>` in each `<form-bean>` and in each `<form-property>`. We also need to call `addSetProperties()` twice, to populate the `Parameter` object with the property/value pair. Finally, we must call `addSetNext()` twice so that `addParameter` is called on `Bean` and `Property`.

7. Change the `protected` access of `DynaFormsLoaderFactory.instantiate()` to `public` access. We'll use this function to instantiate custom subclasses of `FormBeanConfig` and `FormPropertyConfig`.

8. Modify `getFormBeanConfig()` on the `Bean` class and `getFormPropertyConfig()` on the `Property` class so that they return the correct `FormBeanConfig`/`FormPropertyConfig` subclass, based on the value of the `className` variable of step 1. Use the `DynaFormsLoaderFactory.instantiate()` function to do this.

9. At this point, the returned `config` objects of step 8 are unconfigured, since we haven't passed them the information from `<set-property>`s. You can use the Apache Commons Bean framework to do this. Like Digester, the JAR file (`commons-beanutils.jar`) comes bundled with Struts. The class you want to use is `org.apache.commons.beanutils.BeanUtils`. To populate the custom `config` subclass with the information from the `_properties` `HashMap` of step 2, you call `BeanUtils.populate(config, _properties)`, where `config` is the custom `config` subclass of step 8. Make this change to both `getFormBeanConfig()` on the `Bean` class and `getFormPropertyConfig()` on the `Property` class so that the returned `config` object is properly configured.

As usual, compile and test your work with your own custom `FormBeanConfig` or `FormPropertyConfig` classes, each with (for example) a `message` property. (Remember to make appropriate `<set-property>` additions to your form bean declaration file.) You can get the message to be displayed when the `getMessage()` function is called.

Useful Links

- "Adding Spice to Struts," parts 1 and 2 by Samudra Gupta: `http://javaboutique.internet.com/tutorials/Dynaform/` and `http://javaboutique.internet.com/tutorials/Dynaform/index-6.html`

- A good article explaining how to use the Apache Digester framework, by Keld H. Hansen: `http://javaboutique.internet.com/tutorials/digester/`

- Eclipse website: `www.eclipse.org`

- The Struts website: `http://struts.apache.org/acquiring.html#Source_Code`

Summary

- Plug-ins, first introduced in Struts 1.1, are a good way to extend or amend the functionality of Struts.
- A plug-in has to implement the `org.apache.struts.action.PlugIn` interface.
- This interface has two functions: `init()`, which is called when the `PlugIn` implementation is instantiated, and `destroy()`, which is called when Struts shuts down.
- Additional parameters may be passed to the plug-in before initialization using `<set-property>` tags in the plug-in declaration. Each `<set-property>` must have a corresponding `setXXX` defined on the plug-in implementation.

CHAPTER 20

JavaServer Faces and Struts Shale

In this chapter, we'll take a look at a web application technology called JavaServer Faces (JSF) that will play a huge role in the Java webapp framework scene for many years to come.

We'll also preview a newer technology called Struts Shale, which apart from the common moniker, shares little with Struts as you know it. Struts Shale, or Shale, for short, is the first among many new webapp frameworks that have a Struts heritage but embrace the advantages of JSF.

Besides these two, we'll also give you a peek at the Struts-Faces integration library, which allows you to use JSF alongside "classic" Struts (that is, Struts as described in this book).

Before we delve into the exciting details, I'd like to give you a bird's-eye view of the two main players in this chapter, JSF and Shale.

JSF Overview

JSF is a webapp framework whose focus is the View tier of the MVC design pattern.

In fact, JSF's *primary goal* is to provide a *standard architecture* for *web-based user interface (UI) components*.

That's quite a mouthful, and it might help to tackle the key words one at a time:

- **Primary goal**: Because it is a fully functional webapp framework, JSF does a bunch of other things apart from the View tier, like control flow and validation. But don't be fooled—it's primarily about the View tier. I believe having this zeitgeist at the outset helps you understand JSF better. JSF and Struts are quite different things, and you would do well to avoid making premature comparisons between them.

- **Standard**: JSF is *not* software. It is a specification (like the Servlet specification or the specification for JSP) that anyone can use to create an actual implementation. Two are available now: the JSF reference implementation from Sun, and Apache's MyFaces, which we'll use in this chapter.

- **Architecture**: The emphasis of JSF is on architecture—how the various components talk among themselves, how requests are processed, and so forth. The emphasis is not on specific components (like a drop-down list, a tab view, or a validator for email addresses). In fact, JSF is specifically designed to be extended.

- **Web-based UI components**: An example might help. I use OpenOffice as my word processor. I've customized the toolbar so that it displays certain buttons every time I load OpenOffice. Some buttons are highlighted, like the Align button to indicate the currently active alignment. Also, when I click Save As, OpenOffice remembers the directory I last saved a file in and gives me a Save dialog box with that folder selected. Now imagine implementing all these features in your own webapp. Where would you start? The reality is that making a web-based application behave like an ordinary desktop app is very challenging. Web-based UI components and the request processing machinery behind them help you achieve this goal.

Unlike Struts, which usually gives you just one or two ways to accomplish a given task, the JSF specification is very flexible, and there are many combinations you can pick from to accomplish a task. This flexibility might create the false impression that JSF is too complex to work with.

In one sense this is true. JSF is complex, but most of the complexity is hidden from page developers and application developers who use it to build webapps. The complexity only becomes apparent when you're extending JSF—for example, by creating new UI components, or providing integration points to other frameworks (e.g., the Commons Validator framework or Spring).

In short, it's quite easy to *use* JSF to build webapps, but it's more difficult to make significant extensions to it. And among the possible ways to extend JSF, it's probably easiest to make extensions to the UI components and validators.

Shale Overview

Unlike JSF, which is a specification, Shale is actual software that you can use immediately.

Shale is touted to be a complete reworking of Struts (which we'll sometimes refer to as "classic" Struts for clarity). The reworking has been quite thorough, and it's unlikely that a Struts developer will recognize it as Struts!

Shale is based on JSF. By this, I mean JSF is to Shale what JSP is to Struts. There would be no Shale without JSF. You need an implementation of JSF (like MyFaces) in order for Shale to work. I don't mean to belabor the point, but it is crucial that you understand this.

STRUTS TI AND STRUTS OVERDRIVE

Struts Shale isn't the only "nouveau Struts" around—"Struts Ti" and "Struts OverDrive" are both under development.

Struts Ti (short for Titanium) is a merging of classic Struts and WebWork. WebWork is an open source web application framework, generally viewed as being technically sounder than Struts, yet similar enough to it for a merging to be possible. At the time of this writing, information on Struts OverDrive is scarce, but I'm guessing that it's Struts using the Nexus framework for View/Controller communication. The Nexus framework handles text formatting, localization, and type conversions. This framework actually applies to many different webapp frameworks (Spring, Struts, etc.) and presentation layers (JSP, ASP.NET, etc.). OverDrive is the brand name given to applications that utilize the Nexus framework. Refer to "Useful Links" for details on both Ti and OverDrive.

As with Shale, these projects are quite different compared to Struts as you know it from this book. In my (admittedly cynical) view, they have a "Struts" prefix to tap into the large existing Struts user base. But I fear this is a mistake. Splitting a brand name only makes sense if there are distinct, clearly identifiable niches that each split addresses. Microsoft's desktop/server offerings are a good example. Each addresses a clearly different niche under the "Windows" umbrella. But in the case of webapp frameworks, no clear multiplicity of niches is apparent, so having *four* frameworks called "Struts" simply creates confusion, actually diluting the Struts brand. I believe most CIOs would stick with the tried-and-tested classic Struts, or if they were intending to migrate, consider more mature frameworks like Spring or standards-based JSF. Very few would explore Struts X. In fact, it is more likely that they would have chosen the original WebWork instead of its morph with Struts (that is, Struts Ti). Now, they are likely to look at neither one.

In my opinion, it's the project that remains backward compatible with Struts 1.*x*, while introducing major improvements over time, that will win the day—or rather, the users. At the time of this writing, that winning horse looks like Struts 1.*x*. This likelihood is increased if the Struts-Faces integration library (described later in this chapter) sees more development. Shale might come in a close second because of the superior features it offers.

Shale has three major design objectives:

- **Expanding JSF past the View tier**: Shale provides support for areas poorly addressed by the JSF specification (like the practicalities of validating user input). Recall that JSF is mainly about the View tier, so Shale has to do a fair bit to make JSF palatable to longtime Struts developers used to the comforts of the Validator framework, dynamic forms, and Tiles.

- **Adding features previously unavailable in Struts**: For example, Struts does not address user interactions spanning multiple page requests. Shale has a feature called "dialogs" (borrowed from Spring's "webflow") to do this.

- **Integration with other webapp frameworks**: Primarily the Spring framework. There is also some support for Ajax technologies.

So beware! Shale is *not* a natural evolution of "classic" Struts—you can't just "upgrade" your Struts webapp to Shale.

So why the "Struts" prefix? The reason is probably because the most useful parts of Struts have been successfully refactored into Apache Commons projects (the Validator framework and dynamic forms), or are largely independent of Struts (like Tiles is). Shale reuses these to provide important conveniences not present in the JSF specification. In this sense, classic Struts is part of Shale. Note also a more cynical alternative we've discussed in the sidebar, "Struts Ti and Struts OverDrive."

Learning Struts a Waste of Time?

The preceding introduction might give you pause for thought: is learning Struts a waste of time?

By no means!

"Classic" Struts is by far the most popular webapp framework to date, and will very likely coexist with both JSF and Shale for some time yet. As you'll see later in this chapter, there are a number of areas (like validation and dynamic forms) that JSF does not address as well as Struts. This was intentional, since JSF focuses on the View tier, and leaves it to other technologies to addresses its deficiencies in other areas. This in fact is the raison d'être for Shale. Unfortunately, at the time of this writing, Shale is just at version 1.0, nowhere near the maturity of classic Struts.

Also, classic Struts is still under active development, so new features and improvements are likely, further spurring its adoption by those who do not need the advanced features of JSF or Shale.

Lastly, a good working knowledge of Struts gives you an edge in picking up the newer technologies, and gives you a better vantage point from which to evaluate their effectiveness for your needs.

JavaServer Faces (JSF)

As you might gather by now, JSF focuses on the View tier. Viewed from a Struts developer's eyes, this focus implies two important differences between JSF and Struts.

- **A more complex request processing lifecycle**: Because JSF uses server-side UI components, it processes incoming HTTP requests differently from Struts. JSF's request processing lifecycle is divided into several *phases*. This is unlike Struts, which has just two phases: one for when the `ActionForm` gets validated and the other when the `Action`'s `execute()` function is called. Also, JSF allows your own classes to listen for *events* at the end of each phase (more on this in the following subsections). This is a huge improvement over Struts, and is what makes JSF so extensible.

- **Some configuration information moved away from configuration files into JSPs**: JSF has no notion of the Validation framework, dynamic forms, or Tiles, so config files for these do not exist. Instead, validations, for example (and as you will see, many other things), are specified in JSPs that make up the webapp. Like Struts, JSF does have a config file, called `faces-config.xml`, so not all the configuration information goes out to JSPs.

It is important to take note of these two points to help lessen the cognitive dissonance you (as a Struts developer) might experience when learning JSF.

Server-Side UI Components

When a page is submitted to a Struts webapp, the form data on the page is stored in an `ActionForm` subclass. Normally, we'd think of the `ActionForm` as a temporary storage area for data destined for the Model tier.

While perfectly true, another way of looking at the `ActionForm` is to say that it stores the *state* of the View tier on the server. This state information can be used to redisplay the page if necessary (e.g., when you have simple validation errors) in its original condition.

For example, in the Registration webapp (see Chapter 5), when the user keys in an incorrect password, the page is redisplayed with the user ID that was originally keyed in.

With our new way of seeing things, we'd say that the data saved on the `ActionForm` was used to put the state of the View (the original user ID redisplayed) back into its original state.

JSF takes this way of seeing things to the extreme. JSF has tags for various UI components just like Struts has, with its `<html:xxx>` and `<bean:xxx>` tags. The big difference is that *each JSF tag causes an object to be created on the server*. This object is used to hold the UI component's state. We'll elaborate on this shortly.

One other thing that JSF takes to the extreme is the organization of the JSF tags. In Struts, we're accustomed to a simple two-level nested hierarchy: an `<html:form>` can contain elements like an `<html:input>` or `<html:submit>`, as in this snippet:

```
<html:form action="MyHandler.do">
  <bean:message key="myapp.prompt.name"/></td>
  <html:text property="name" size="60" />
  <html:errors property="name"/>
  <html:submit>
     <bean:message key="myapp.prompt.submit"/>
  </html:submit>
</html:form>
```

This is a reflection of the fact that Struts display tags closely mirror ordinary HTML tags. The hierarchy for JSF, on the other hand, is much richer and you typically have more levels, as shown in Listing 20-1.

Listing 20-1. *A Snippet of JSF Tags Showing a Hierarchical Organization*

```
<f:view>
    <h:panelGroup>
        <h:messages style="color:red" globalOnly="true"/>
        <h:outputText value="#{reg_messages['title_reg']}" />
        <h:form binding="#{user.container}">
            <h:outputText value="#{reg_messages['user_id']}"/>
              <h:inputText id="userId"
                      value="#{user.userId}" required="true">
               <x:validateRegExpr pattern='^[A-Za-z0-9]{5,10}$' />
            </h:inputText>
            <h:message for="userId" />
...
```

This hierarchical organization (the hierarchy is defined through nesting) allows you to finely control how the page is rendered for display to the user. You'll see why later in this chapter.

For now, the important thing to note is that the corresponding server-side objects created by the page in Listing 20-1 are also nested in a tree-like fashion, as shown in Figure 20-1.

The first time a JSF page is requested, the tree of objects—let's call it the "UI tree"—is created on the server. JSF uses the UI tree to create the HTML for the requested page.

The *next* time the same page is requested (by the same user, in the same session), this UI tree is re-created on the server, and is populated with data from the page. This data does not have to be only user input destined for the Model tier but can also include the states of each UI component.

For example, suppose a JSF page contains a tag for a treeview UI component, as you might have used in Microsoft Explorer or Outlook. A user requesting the page sees the treeview and clicks a node on it. The treeview expands, and the user clicks further, exposing more of the tree, as in Figure 20-2.

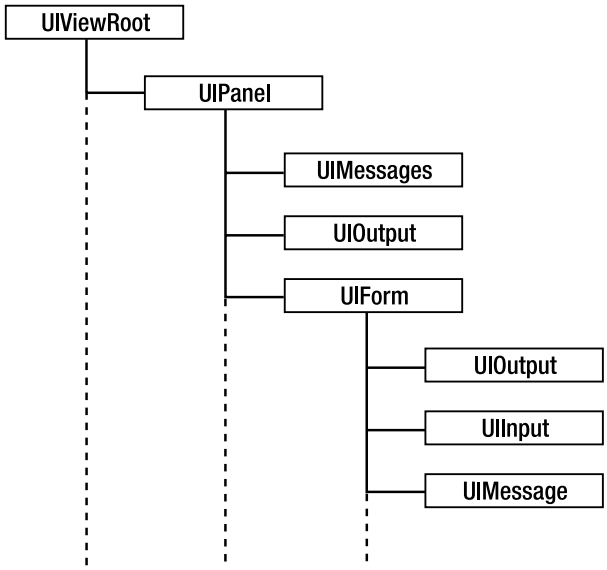

Figure 20-1. *Corresponding UI tree on the server*

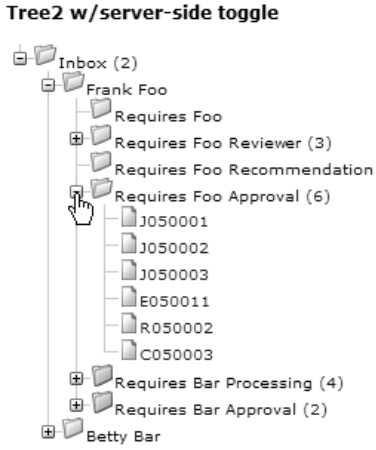

Figure 20-2. *A treeview in MyFaces*

When the page is submitted, the state of the treeview—in other words, which nodes were expanded by the user—would be sent unobtrusively along with other data on the form. This data is stored on the server-side object representing the treeview component.

To summarize, unlike Struts, where a page's data goes into a single `ActionForm`, in JSF a server-side UI *tree* is created to hold data—both the data destined for the Model tier and the data representing the state of the View. JSF makes no distinction between the two.

So, it is the UI tree that needs to be processed, just as Struts processes `ActionForms`. The user-submitted data contained in the UI tree needs to be validated and processed for delivery into the Model tier. We'll see how this is done next.

Request Processing Lifecycle

JSF's request processing lifecycle describes how HTTP requests from a client are processed to generate a response by the server. JSF's request processing lifecycle is divided into several *phases*. Figure 20-3 illustrates this.

■ **Tip** You need to know the material in this section very well, because the available documentation on JSF and documentation for technologies based on JSF, like Shale, frequently refer to the various phases of JSF's request processing lifecycle.

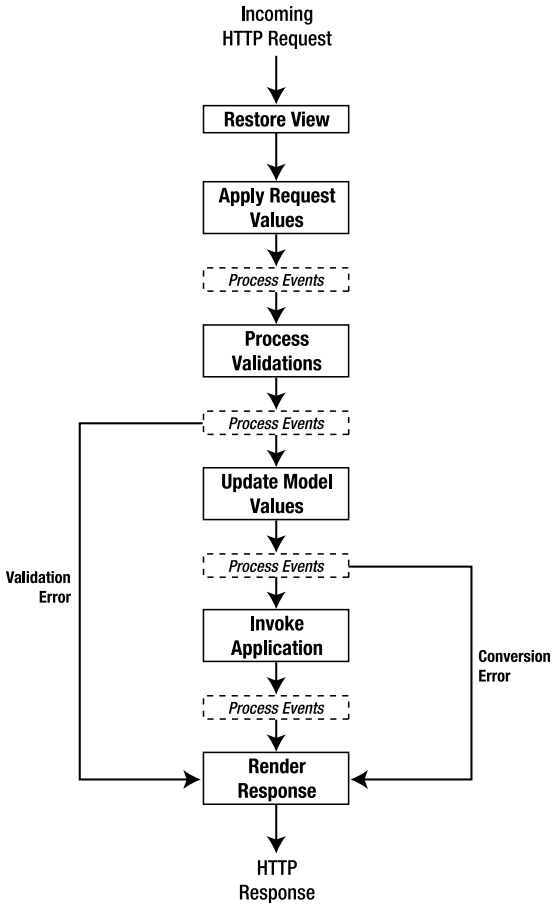

Figure 20-3. *JSF's request processing lifecycle*

There are six phases in all (the solid boxes in Figure 20-3), and each phase and the *events* associated with it are completely processed before the subsequent phase is started.

■ **Note** *Events* and *event listeners* are the mechanism by which JSF exposes its internals to third-party classes. They are an important extension point of JSF. We'll describe both of these in the following section.

In Figure 20-3, we've marked two ways (validation errors and conversion errors) by which processing gets fast-tracked to the `Render Response` phase. In truth, event listeners that process events immediately after *any* phase may cause JSF to move into the `Render Response` phase.

FACES REQUESTS

Not all HTTP requests sent to the server are processed according to Figure 20-3, but only those requesting a JSF page. Such requests are termed *faces requests*. This is analogous to the way not every HTTP request is processed by Struts, but only those with the `.do` extension (by default). The corresponding extension for JSF is `.jsf`. (Again, this depends on how you've configured `web.xml`. Another common choice for a JSF extension is `.faces`.)

Your pages need not have the extension `.jsf`, but the call to them must. For example, suppose you have a JSP page containing JSF markup named `mypage.jsp`. To correctly call this page, you'd have to make a request for

```
http://www.mycompany.com/mypage.jsf
```

If you were to make a request for `http://www.mycompany.com/mypage.jsp` directly, you'd get an error message.

You can name your pages with a `.jsf` extension, but this would *not* work with a JSP 1.2–compliant servlet container. JSP 1.2 requires JSP pages to have the `.jsp` extension. So, the most portable method is to name your JSF pages `.jsp` and request them as `.jsf`.

Let's look at each phase in detail. Don't worry if some of my comments seem cryptic at the moment. I'll explain the cryptic bits in the following subsections.

- **Restore View**: If this is the first time the user requests this page, then the *UI tree* (see the discussion in the previous subsection) is created. If this is the second time the page is created, then the UI tree is re-created with the previously saved state. *Value bindings* for each UI component are also processed at this phase.

- **Apply Request Values**: Each UI component on the UI tree is given a chance to update its state based on the information contained in the incoming HTTP request.
- **Process Validations**: In this phase, validations are processed. If there are errors, then processing is taken directly to the Render Response phase.
- **Update Model Values**: As its name implies, in this phase the Model tier is updated with the data from the UI tree. JSF knows which UI components hold data destined for the Model tier because you've specified this in the JSF markup. Data conversion errors will cause processing to proceed directly to the Render Response phase.
- **Invoke Application**: Navigation is performed based on *action events* posted during previous phases. It is also possible for your customizations of JSF to add other processing code to this phase.
- **Render Response**: The HTML output is created, based on either the navigation results or the existing UI tree (if the page is to be redisplayed). The state of the UI tree is also saved for processing on subsequent requests.

After every phase (except for Restore View and Render Response), events that might have been generated during that phase are processed.

Events are Java objects that contain information indicating a certain processing state. Events are generated by UI components on the UI tree, or by your own processing code attached to any phase. Processing of events is done either by JSF itself or by event listeners. We'll see how this is done next.

Events and Event Listeners

In this book, you've come across two design patterns: the MVC design pattern and the Singleton design pattern (in Chapter 19). Events and event listeners fall under a simple but powerful design pattern called *Observer*.

In the Observer design pattern, there are two roles. The first is that of a Publisher, a data source that gets updated from time to time. The second role is that of Subscriber, denoting entities interested in being updated with the latest data available on the Publisher.

A bad way to get Subscribers updated with the latest data is for them to interrogate the Publisher at regular intervals. This technique, called polling, is obviously inefficient: poll too fast and you waste CPU time; poll too slowly and you risk getting the data too late or losing it altogether.

Instead of polling, the Observer design pattern employs the "don't call us, we'll call you" maxim. Subscribers interested in getting data from the Publisher first *register* with the Publisher. When the new data arrives, the Publisher *informs* all its registered Subscribers.

Let's take a concrete example. GUI frameworks like Swing or Abstract Windowing Toolkit (AWT) have classes representing visually displayed UI components, like buttons, check boxes, and so forth. Each of these GUI components needs to signal certain user interactions, like a mouse click over a button, or a key pressed in a text input box. Each such activity is called an *event*.

An event is usually represented by a Java class, which has fields containing data describing the event. For example, in the case of mouse clicks, a `MouseClickEvent` might contain the X and Y coordinates of the mouse click, as shown in Listing 20-2.

Listing 20-2. *A Class to Represent Mouse Clicks*

```
public class MouseClickEvent{

   private int _x, _y;

   public  MouseClickEvent(int x, int y){
      _x = x; _y = y;
   }

   public int getX(){ return _x; }

   public int getY(){ return _y; }
}
```

Notice that classes representing events are usually *immutable*, meaning that you can only read data from them once they are created. The reason for this is because an event could be shared among many Subscribers. You don't want any Subscriber to amend the event.

So we now have a class to represent mouse clicks. Great! But how to guarantee that Subscribers can actually understand the data contained in this event? Simple—have them implement a standard interface, shown in Listing 20-3.

Listing 20-3. *The MouseClickListener Interface*

```
public interface MouseClickListener{

   public mouseClicked(MouseClickEvent mce);

}
```

So, a Subscriber implementing `MouseClickListener` explicitly declares that it understands how to process mouse clicks. Listing 20-4 shows an example of a Subscriber implementing `MouseClickListener`.

Listing 20-4. *Example of Subscriber Implementing MouseClickListener*

```
public class MySubscriber implements MouseClickListener{

   public mouseClicked(MouseClickEvent mce){
     System.out.println("x coordinate = " + mce.getX());
     System.out.println("y coordinate = " + mce.getY());
   }

}
```

What about Publishers? Do they have to implement any interface? Usually not, but Publishers should have a function to add or remove event listeners. Subscribers would have to know these functions and call them; an example is shown in Listing 20-5.

Listing 20-5. *Fragment of a GUI Widget That Generates Mouse Click Events*

```
public class MyGuiWidget{

 private Set _mouseClickListeners = new HashSet();

 public  addMouseClickListener(MouseClickListener mcl){
   _mouseClickListeners.add(mcl);
 }

 public  removeMouseClickListener(MouseClickListener mcl){
   _mouseClickListeners.remove(mcl);
 }

 protected void broadcastMouseClick(int x, int y){
   if(_mouseClickListeners.isEmpty()) return;

   MouseClickEvent mce = new MouseClickEvent(x,y);

   for(Iterator listeners = _mouseClickListeners.iterator();
       listeners.hasNext();){

       ((MouseClickListener) listeners.next()).mouseClicked(mce);
   }
 }
```

```
 // other functions that receive mouse clicks from the screen
 // and then call broadcastMouseClick()
 ...

}
```

IS MOUSECLICKEVENT REALLY NECESSARY?

You might realize by now that the `MouseClickEvent` class is not strictly necessary. In fact, it's possible to eliminate this class completely, and instead put in the X and Y coordinates as arguments of `mouseClicked()`:

```
public interface MouseClickListener{

   public mouseClicked(int x, int y);

}
```

Both approaches are valid, and JSF uses both, but most often the first.

Your Java application would hook up `MyGuiWidget` and `MySubscriber(s)` as shown in Listing 20-6.

Listing 20-6. *Fragment of a GUI Widget That Generates Mouse Click Events*

```
MyGuiWidget widget = new MyGuiWidget();

MySubscriber sub1 = new MySubscriber();
MySubscriber sub2 = new MySubscriber();

widget.addMouseClickListener(sub1);
widget.addMouseClickListener(sub2);
```

Now that you understand all this, you might be wondering how it relates to JSF. Well, at each phase (except Restore View and Render Response), events would be generated. These events might relate to the change in phase (`PhaseEvent`) or indicate that the user requires a certain action to be performed, such as navigating to a next page or calling a function on one of your classes (`ActionEvent`) or signaling that a UI component's value has changed (`ValueChangedEvent`).

Also, as the sidebar points out, it's possible to flag events without having an explicit "event" object. Validations and Model tier updates are done this way. Essentially, you

create listeners for each of these and somehow (you'll see how shortly) register them with JSF. JSF will call these listeners at the appropriate phases.

So, the million-dollar question is how do we register listeners for the various events? In Listing 20-6, you can see how this might be done programmatically using the Java language. This is *not* the norm for JSF. Instead, you hook up the event producer with the event listener *in the JSP page* that specifies the UI tree.

We'll postpone giving you an example of how this might be done until after we've addressed a few more loose ends.

JSF Tag Libraries

There are just two standard JSF tag libraries:

- The Core tag library deals with JSF functionality that is independent of the way the View is rendered. Remember, JSF is very flexible, and it need not work with just HTML as its view medium. It's possible to use JSF to render to Wireless Markup Language (WML), for example. The Core taglib contains, for example, tags to run validations or tags to load message resources.

- The HTML tag library defines tags specifically describing the View tier, specifically for HTML output.

I'd like to remind you that the set of "standard" UI components and validators is fairly limited. To create complex webapps, you will likely want to extend JSF. There are two ways to do this:

- Use third-party UI components and validators. MyFaces comes with a few of these.

- Use a framework that specifically addresses deficiencies in JSF. Shale is one such framework.

Value and Method Binding

JSF introduces two very powerful techniques called *value* and *method binding*.

You've had some experience with value binding in Struts. Remember how you could read and write properties from your form bean in your Struts tags? For example, if you wanted an `<html:text>` to save a property called `userId` to a form bean called `user`, you'd do this:

```
<html:text property="userId"/>
```

where the `user` form bean is implicit. That's what value binding is about. However, the JSF syntax is different:

```
<h:inputText value="#{user.userId}"/>
```

JSF uses an expression language syntax (see Chapter 10). Note that this is not the same as the JSTL "EL" syntax, since it begins with a hash symbol (#).

Method binding is an extension of this idea. It allows your JSF tags to call functions on your form bean (the JSF term is *backing bean*). For example:

```
<h:commandButton action="#{user.Logon}" />
```

describes a button that, if clicked, calls the `Logon()` function on the `user` backing bean.

This technique is powerful because you can define multiple buttons on a single form, and bind them to different functions, each to process the form data differently.

Quick Quiz

How would you do the same thing in Struts?

Because of this, backing beans are sometimes like `ActionForm` and `Action` rolled into one.

When you read the JSF literature, you will certainly come across the term *managed bean*. A managed bean is a backing bean whose instantiation is controlled by JSF.

This is analogous to form beans in Struts. The common characteristic of form beans (in Struts) and managed beans (JSF) is that the instantiation of both is performed by their respective frameworks.

The declaration of a managed bean in `faces-config.xml` (the JSF config file) is also different. Instead of

```
<form-bean name="user" type="net.thinksquared.reg.User" />
```

you'd say

```
<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>net.thinksquared.reg.User</managed-bean-class>
</managed-bean>
```

The JSF style is more verbose because rather than using attributes, it uses separate tags.

Navigation

In Struts, navigation information (`<forward>`s) is mixed with processing declarations (`<action>`s). JSF does a much better job at representing navigation information.

In fact, the whole JSF navigation model is much easier to understand and use compared to Struts. Instead of a specialized class like `ActionForward`, JSF uses plain Java `String`s.

So, instead of returning the equivalent of `ActionForward`, the functions on your backing beans that perform navigation would return a `String`. These are called *logical outcomes*. As an example, consider Listing 20-7.

Listing 20-7. *A Simple Navigation Rule*

```
<navigation-rule>
  <from-view-id>/index.jsp</from-view-id>
  <navigation-case>
    <from-outcome>register </from-outcome>
    <to-view-id>/register.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/success.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

This example defines a single *navigation rule*, with two logical outcomes: `register` and `success`. You can define as many navigation rules as you wish, and even add conditions like

```
<from-view-id>/index.jsp</from-view-id>
```

which tells JSF what the "previous" page must be in order for this `<navigation-rule>` to apply.

Example: The Registration Webapp

In this section, we'll port our old friend the Registration webapp (see Chapter 5) to JSF. First, here are the specs:

- There are three pages: one to perform user logon, one for user registration, and the last to indicate successful logon *or* registration.
- The logon page includes the usual form containing user ID and password fields. The page also contains a link to the registration page. If the system detects that the user ID entered doesn't exist, it forwards to the registration page.
- The registration page is similar to the logon page, except that it contains an extra "confirm password" field.
- As usual, the fields are validated before the input is accepted for delivery to the Model tier.

■ **Caution** This example does *not* show you the full power of JSF, nor does it show you how to use it efficiently. We're using the Registration webapp simply because we've implemented it with Struts and Tiles.

I hope this gives you some mental scaffolding with which to pick up the new concepts in JSF. We'll start with configuring JSF.

Configuring JSF

As with Struts, there are two files to use when configuring JSF: `web.xml`, the servlet configuration file, and `faces-config.xml`. Both of these should be in the `WEB-INF` directory.

The `web.xml` file contains information telling the servlet container what request extensions (like `.jsf`) should be shuttled to JSF, and where to store state (client or server). A `web.xml` file with a default setting is available in the MyFaces distribution. You should be able to use this file without any amendment.

The `faces-config.xml` file contains information declaring the `<managed-bean>`, `<navigation-rule>`, and other tags to plug in third-party UI components and validators, among other things. Listing 20-8 shows the `faces-config.xml` file for the Registration webapp.

Listing 20-8. *faces-config.xml for the Registration Webapp*

```
<?xml version="1.0"?>

<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>

  <managed-bean>
    <managed-bean-name>user</managed-bean-name>
    <managed-bean-class>net.thinksquared.reg.User</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>

  <navigation-rule>
    <navigation-case>
      <from-outcome>register</from-outcome>
      <to-view-id>/register.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
```

```
      <from-outcome>success</from-outcome>
      <to-view-id>/success.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

</faces-config>
```

We've declared a single managed bean called `user`. This bean both holds the user ID and password data *and* performs complex validation. Note that the `user` bean is declared to have request scope.

We've also declared a single navigation rule, mapping the registration page (`register.jsp`) to the logical outcome `register` and, similarly, the page displaying successful registration (`success.jsp`) to the logical outcome `success`.

Message Resources

Like Struts, JSF also supports internationalization by using message resource bundles. JSF's support is much better than Struts. Instead of specifying the `application.properties` file in the configuration file, we can declare it in a JSP that uses the message resource.

This means that we can use multiple message resources. You'll see this in action shortly. Listing 20-9 shows the message resource for the Registration webapp. The file is stored as `net\thinksquared\reg\messages.properties`, and will therefore be referred to as `net.thinksquared.reg.messages`.

Listing 20-9. *Message Resource File for the Registration Webapp*

```
#-------- error prompts --------
user_exists         = The userid {0} is taken. Please try another.
user_not_registered = You have not registered! Please do so now.
wrong_password      = Incorrect password. Please try again.
password_mismatch   = The passwords you keyed in don't match.

#-------- button text ----------
register  = Register
logon     = Logon

#-------- other text  ----------
title         = Please log on
title_reg     = Please register
new_user      = Register as new user...
registered_ok = You''ve registered as {0} with the password {1}!
```

```
logon_ok      = You''re logged on as {0}
user_id       = User ID:
password      = Password:
password2     = Re-type password:
```

Notice that the message for `logon_ok` begins with `You''re`. The repeated single quotes are not a mistake—it's the way to "escape" the single quote.

The User Backing Bean

The `user` backing bean (which is also a managed bean) is used to store data (the user ID and password) and also has functions to log on an existing user or register a new user, as shown in Listing 20-10.

Listing 20-10. *The user Backing Bean*

```
package net.thinksquared.reg;

import java.util.Map;
import java.util.HashMap;
import java.util.ResourceBundle;
import java.text.MessageFormat;

import javax.faces.context.FacesContext;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;

public class User {

   private static Map _dbase = new HashMap(); //our dummy database.

   private String _uid,_pwd;

   private UIComponent _container;

   public User(){
      _uid = null;
      _pwd = null;
      _container = null;
   }
```

```
//------------------------------------ data get/set

public void setUserId(String uid){
   _uid = uid;
}

public String getUserId(){
   return _uid;
}

public void setPassword(String pwd){
   _pwd = pwd;
}

public String getPassword(){
   return _pwd;
}

//------------------------------------ UI get/set

public void setContainer(UIComponent container){
   _container = container;
}
public UIComponent getContainer(){
   //we must return null,otherwise the
   //saved _container will be grafted
   //into the new UI tree.
   return null;
}

//------------------------------------ Actions

public String Register(){

   //get a handle to the current "context"
   FacesContext context = FacesContext.getCurrentInstance();

   //get a handle to the desired properties file
   ResourceBundle bundle =
      ResourceBundle.getBundle("net.thinksquared.reg.messages",
         context.getViewRoot().getLocale());
```

```
    //complex validation: user exists?
    if(_dbase.get(_uid) != null){

       Object[] params = {_uid};
       String msg = MessageFormat.format(
          bundle.getString("user_exists"),params);

       String clientId =
          _container.findComponent("userId").getClientId(context);

       context.addMessage(clientId, new FacesMessage(msg));

       //returning null causes input page to be re-displayed
       return null;
    }

    //everything OK - go ahead and register the user
    _dbase.put(_uid,_pwd);

    Object[] params = {_uid,_pwd};
    String msg = MessageFormat.format(
       bundle.getString("registered_ok"),params);

    context.addMessage (null, new FacesMessage(msg));
    return "success";
}

public String Logon(){

    //get a handle to the current "context"
    FacesContext context = FacesContext.getCurrentInstance();

    //get a handle to the desired properties file
    ResourceBundle bundle =
       ResourceBundle.getBundle("net.thinksquared.reg.messages",
          context.getViewRoot().getLocale());

    //complex validations:
    //  does the user exist?
    //  does the given pwd match the one in database?
    Object pwd = _dbase.get(_uid);
    if(null == pwd){
```

```
            String msg = bundle.getString("user_not_registered");
            context.addMessage (null, new FacesMessage(msg));
            return "register";

        }else if(!pwd.equals(_pwd)){

            String msg = bundle.getString("wrong_password");
            String clientId =
               _container.findComponent("password").getClientId(context);

            context.addMessage(clientId, new FacesMessage(msg));
            return null;
        }

        //everything OK - go ahead and log in the user
        //code to log on user here...
        Object[] params = {_uid};
        String msg = MessageFormat.format(
            bundle.getString("logon_ok"),params);

        context.addMessage (null, new FacesMessage(msg));
        return "success";
    }
}
```

There are three sections in Listing 20-10. The first is a set of getters and setters for data. JSF will call these functions during the Update Model Values phase. JSF knows it needs to call these functions because we've used value binding to link the input fields to these functions. (See Listings 20-11 and 20-12. The value bindings appear as `value=#{...}` on the input fields.) This is similar to how you'd link Struts tag inputs to `ActionForms`.

The second section consists of a single getter and setter for a *UI component*. This should be entirely new to you since Struts does not have UI components. We've bound the `<h:form>` UI component to the `user` bean. (See Listings 20-11 and 20-12. This value binding appears as `binding=#{...}` on the `<h:form>` tag.) The reason for wanting to have a copy of the UI component corresponding to `<h:form>` will become obvious shortly.

The last section has two functions: one to register new users and another to log on existing users. We'll describe the `Register()` function only, since there are no new technicalities in `Logon()`.

First, notice that the return value for `Register()` is a `String`. This string represents the *logical outcome* (representing the "next" page) associated with a `<navigation-case>` (refer to Listing 20-8).

The first thing we do in `Register()` is get a reference to a `FacesContext` object:

```
FacesContext context = FacesContext.getCurrentInstance();
```

This `FacesContext` object holds the root node of the UI tree and exposes a variety of useful functions.

The next step is to get a message resource bundle so we can output messages to the View:

```
//get a handle to the desired properties file
ResourceBundle bundle =
    ResourceBundle.getBundle("net.thinksquared.reg.messages",
        context.getViewRoot().getLocale());
```

This is more unwieldy compared to Struts because JSF allows us to have multiple message resource files. You have to locate these files by name *and* from the right locale. The statement

```
context.getViewRoot().getLocale()
```

obviously gets the current request's locale. Note that `getViewRoot()` is a function on `FacesContext` that returns the root node of the UI tree.

Once we have handles to a `FacesContext` and the right `ResourceBundle`, we can begin complex validation:

```
//complex validation: user exists?
if(_dbase.get(_uid) != null){

   Object[] params = {_uid};
   String msg = MessageFormat.format(
                       bundle.getString("user_exists"),params);

   String clientId =
              _container.findComponent("userId").getClientId(context);

   context.addMessage(clientId, new FacesMessage(msg));

   //returning null causes input page to be re-displayed
   return null;
}
```

The only bit of code to perform complex validation (checking that the user ID doesn't already exist) is in the clause of the `if(...)` statement. The body of the `if()` statement deals getting the error message to display in the right place! This might seem a little daunting at first, so let's take it one step at a time. The error message in question is `user_exists`, which the message resource file (see Listing 20-9) tells us is

```
user_exists = The userid {0} is taken. Please try another.
```

As you can see, it contains a single replacement variable, {0}, which we want to replace with the user ID that the user keyed in. For example, if the user had keyed in the user ID "Kermit", we want the final error message to read

```
The userid Kermit is taken. Please try another.
```

Java has a function called MessageFormat to perform this substitution for you. That's what we're doing in the first two lines:

```
Object[] params = {_uid};
String msg = MessageFormat.format(
                    bundle.getString("user_exists"),params);
```

The function getString() on the ResourceBundle gets the error message corresponding to the given key (user_exists). We then use MessageFormat to replace the {0} with the _uid, the user ID. So, the msg variable holds the final error message. How can we get this to display on the page? That's what the subsequent lines do.

First, we need to know *where* to display the error message. With Struts, this is easy. But in JSF, because there is the possibility that we're using third-party components, it's a little trickier. Understanding why will help you get a better feel for JSF, so we'll make a little digression at this point.

Recall that in JSF, all the information from a page is stored in a UI tree. Each node (instances of javax.faces.component.UIComponent) on the tree may be labeled using an id attribute. Obviously, labels only make sense if they are unique—that's the only way to locate the node you want. The challenge is ensuring uniqueness while at the same time allowing custom components.

Why? Well, a custom component might be composed of many subcomponents, each of which has to be labeled uniquely. With Struts, ensuring uniqueness is easy, because all you have to do is read the JSP for the page, and change the ids if needed. With custom components in JSF, this will not always work, because there isn't a one-to-one correspondence between tags on the JSP and nodes in the UI tree.

An example of this is the treeview component we discussed earlier. The treeview UI component creates new nodes in the UI tree *at runtime*. It might assign these nodes unique labels, and this is where the difficulty comes in—what if the names of these nodes clash with those from another custom component?

JSF solves these issues by using *naming containers*. A naming container is an ordinary node on the UI tree that appends an additional prefix to all the ids of its child nodes. This prefix is determined by JSF, so the final label of a node is indeed unique.

Most of the time, we'll only know a node's *un*prefixed id. In order to search for a particular node using its unprefixed id, we have to either

- Have a reference to the naming container that contains the node
- Have a reference to another child node within the same naming container

We've taken the second route. That's what the `setContainer()` function is for. It allows JSF to save a copy of a UI component—the `<h:form>` UI component—which belongs to the same naming container as the `<h:message>` tags we wish to locate. In Listing 20-10, we've called this saved copy `_container` because `<h:form>` contains all the other UI components we're interested in (see Listing 20-11).

How does JSF know it needs to call this function? If you look at Listing 20-11, the declaration for `<h:form>` is

```
<h:form binding="#{user.container}">
```

This causes the UI component corresponding to `<h:form>` to be saved on the `User` class using `setContainer()`.

Unfortunately, this isn't the end of the story! When one of our pages is reloaded, or if you navigate from one page to another, JSF also grafts the UI component retrieved from the `user` backing bean using `getContainer()`. It grafts the previously stored `<h:form>` UI component and all its child nodes into the place it sees the binding being made.

JSF does this to allow backing beans to alter the appearance of a page. Used correctly, this is a very powerful technique for creating dynamic Views. You can use this technique to add or remove or reposition elements (like toolbars, buttons, or mini-windows) in your webapp.

In our case, though, we don't want this behavior, so we must set the return value of `getContainer()` to `null`. This tells JSF not to graft anything but to call `setContainer()` and save a copy of the `<h:form>` instead.

With all this background, the next two lines should be easy to interpret:

```
String clientId =
          _container.findComponent("userId").getClientId(context);
context.addMessage(clientId, new FacesMessage(msg));
```

The first line uses `_container` to locate the UI component named `userId` using `findComponent()`. `getClientId()` is then called on this UI component in order to get the absolute `id` (with the naming container's prefix added). The next line uses this absolute `id` (`clientId`) to attach an error message for that UI component.

The last thing we do is to return `null`. This indicates to JSF that there is no "next" page—the current page needs to be redisplayed. This is an advantage over Struts, which requires you to declare the "input" page in `struts-config.xml`.

This brings us to an interesting topic: how exactly does JSF know that it should interpret the return value of `Register()` as a "next" page? The answer is the JSF tag that calls `Register()`:

```
<h:commandButton action="#{user.Register}" ...
```

You can see that the button calls `user.Register()`, and interprets it as an "action." Essentially, the UI component corresponding to this `<h:commandButton>` calls `Register()`, then creates an `ActionEvent` (similar to Struts' `ActionForward`). `ActionEvents` are processed during the

Invoke Application phase. An `ActionEvent` with a `null` logical outcome causes the page to be redisplayed.

This section contains a lot of information to take in one go, so don't be discouraged if you're not confident about the details. I recommend you tackle the next section (after a short break!) and then try out the Registration webapp exercise at the end of this section.

The View

The main Views in the Registration webapp are `logon.jsp` (Listing 20-11), to perform user logon, and `register.jsp` (Listing 20-12), to register new users.

Listing 20-11. *logon.jsp*

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://myfaces.apache.org/extensions" prefix="x"%>
<f:loadBundle basename="net.thinksquared.reg.messages"
                  var="reg_messages"/>
<html>
<body>
  <f:view>
    <h:panelGroup>
      <h:messages style="color:red" globalOnly="true"/>
      <h:outputText value="#{reg_messages['title']}"
                    style="font-weight:bold" />

      <h:form binding="#{user.container}">

        <f:verbatim><table><tr><td></f:verbatim>

        <h:outputText value="#{reg_messages['user_id']}"/>

        <f:verbatim></td><td></f:verbatim>

        <h:inputText id="userId" value="#{user.userId}"
               required="true">
          <x:validateRegExpr pattern="^[A-Za-z0-9]{5,10}$" />
        </h:inputText>
        <h:message for="userId" />

        <f:verbatim></td></tr><tr><td></f:verbatim>
```

```
        <h:outputText value="#{reg_messages['password']}"/>

        <f:verbatim></td><td></f:verbatim>

        <h:inputText id="password" value="#{user.password}"
               required="true">
          <x:validateRegExpr pattern="^[A-Za-z0-9\-\_]{5,10}$" />
        </h:inputText>
        <h:message for="password" />

        <f:verbatim></td></tr><tr><td></td><td></f:verbatim>

        <h:commandButton action="#{user.Logon}"
                         value="#{reg_messages['logon']}"/>

        <f:verbatim></td></tr></table><br></f:verbatim>
      </h:form>
    </h:panelGroup>

    <h:commandLink action="register">
      <h:outputText value="#{reg_messages['new_user']}" />
    </h:commandLink>

  </f:view>
</body>
</html>
```

A few notes for you about Listing 20-11:

- `<f:loadBundle>` loads the given message resource bundle and exposes it with the name defined by the `var` attribute.
- `<f:view>` is a mandatory root tag for JSF. There has to be one of these, and it must contain all your other UI-related JSF tags.
- `<h:panelGroup>` acts like a container for one or more JSF tags.
- `<h:messages>` (note the plural) displays error or other messages saved to the associated `FacesContext` using `addMessage()`. In Listing 20-11, the `globalOnly` attribute indicates that only messages saved using the call `context.addMessage(null, mymessage)` are to be displayed.

- `<h:outputText>` displays text. It's like `<bean:write>` or `<bean:message>`, depending on how you assign the `value` attribute. In Listing 20-11, we've only used it to display prompts or labels from the message resource bundle.

- `<h:form>` allows you to submit form data.

- `<f:verbatim>` simply outputs the text in its body. From one point of view, this is a failing of JSF, because the `<f:verbatim>`s can make your JSP very difficult to read (we've stripped them out in Listing 20-12, so you can more easily see the important bits). On the other hand, it might force you to use layout technologies like CSS to handle layout for your webapp. Take a look at the Zen Garden website (see "Useful Links") for a taste of how powerful CSS can be if used to its full potential.

- `<h:inputText>` renders an input text field. The `required=true` indicates that this field is mandatory. An error message will be posted for this UI component if you submit a blank value.

- `<x:validateRegExpr>` is a MyFaces extension to JSF, allowing you to validate the user input based on a regular expression. This is how you define validators for a component—you nest the validator's tag in the tag for the UI component.

- `<h:commandButton>` displays a button. The `action` attribute must either be a `String` representing a logical outcome, or be bound to a function on a class, whose `String` return value is interpreted as a logical outcome.

- `<h:message>` displays the first message associated with the UI component referred to by the `for` attribute. `<h:message>` should be in the same naming container as the component indicated by its `for` attribute.

- `<h:commandLink>` displays a hyperlink with the given outcome as the target.

We've explained the more difficult attributes in the previous section. It shouldn't be too difficult for you as a Struts developer to understand how these tags work.

Next, let's take a look at the `register.jsp` page. The numerous `<f:verbatim>` tags to control layout can make the code difficult to read, so in Listing 20-12, we've stripped them out.

Listing 20-12. *register.jsp with the <f:verbatim> Tags Stripped Out*

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://myfaces.apache.org/extensions" prefix="x"%>
<f:loadBundle basename="net.thinksquared.reg.messages"
                    var="reg_messages"/>
```

```
<html>
<body>
  <f:view>
    <h:panelGroup>
      <h:messages style="color:red" globalOnly="true"/>
      <h:outputText value="#{reg_messages['title_reg']}"
                    style="font-weight:bold" />

      <h:form binding="#{user.container}">

        <h:outputText value="#{reg_messages['user_id']}"/>

        <h:inputText id="userId"
                  value="#{user.userId}" required="true">
          <x:validateRegExpr pattern="^[A-Za-z0-9]{5,10}$" />
        </h:inputText>

        <h:message for="userId" />

        <h:outputText value="#{reg_messages['password']}"/>

        <h:inputText id="password" value="#{user.password}"
               required="true">
          <x:validateRegExpr pattern="^[A-Za-z0-9\-\_]{5,10}$" />
        </h:inputText>

        <h:message for="password" />

        <h:outputText value="#{reg_messages['password2']}"/>

        <h:inputText id="password2" value="" required="true">
          <x:validateEqual for="password" />
        </h:inputText>
        <h:message for="password2"/>

        <h:commandButton action="#{user.Register}"
                         value="#{reg_messages['register']}"/>

      </h:form>
    </h:panelGroup>
  </f:view>
</body>
</html>
```

Listing 20-12 should hold no surprises for you, except for `<x:validateEqual>`, which is a validator tag that tests whether two fields are equal. `<x:validateEqual>` is yet another example of a MyFaces extension to JSF.

The other interesting thing about Listing 20-12 is that the retyped password is never saved to the `user` backing bean. Instead, the value is discarded after the validation for equality is run.

Try It Out!

The Registration webapp is available in the Source Code section of the Apress website at `http://www.apress.com`, in the file `jsf-registration-webapp.zip`. Copy and unzip this to your development folder and test out the webapp. Play with the code by adding new fields, validations, and new functions.

Where to Next?

I hope this short introduction has succeeded in whetting your interest in JSF. Here are a few pointers on where to go next:

- **Download the spec**: The JSF spec is available for download. The spec is quite readable, and I've found it helpful as a reference. However, you should be aware that it's aimed at those wanting to implement JSF, so it's not really an introduction to JSF.

- **Download the source for MyFaces**: I've found this immensely valuable as a guide to the inner workings of JSF. You might hit a slight bump downloading the source, though: at the time of this writing you need a Subversion client (see "Useful Links") to download the source. You also need Ant (again, see "Useful Links") to automatically download the dependencies (library JAR files) for MyFaces. This can be a pain, so we've done this for you. We've put a copy of the MyFaces 1.1.1 source code in the Source Code section of the Apress website. Also on the Apress website in the Source Code section is a copy of Ant, which you must install in order to compile MyFaces. The Ant distribution contains documentation on how to use Ant. Note that the MyFaces Ant build file needs an Internet connection in order to work.

- **Try out the MyFaces examples**: MyFaces provides many useful components and validators. The interested reader is encouraged to download the `myfaces-examples.war` file available at the Apache site (see "Useful Links") and give it a go. We haven't included this in the Source Code section of the Apress website because the MyFaces team is adding interesting new components all the time, and we don't want you to miss out on the latest goodies.

- The JSF central site has many good JSF tutorials. I've also found the JSF FAQ website very helpful. See "Useful Links" for both.

The next section provides a preview of the new Struts-Faces integration library. This library allows you to use JSF with Struts, and lets you gracefully "upgrade" your existing Struts applications.

Lab 20: The Struts-Faces Integration Library

The Struts-Faces integration library ("Useful Links" tells you where to download the latest distribution) has been around for some time, but it's only recently seen active development. It's essentially a separate JAR file that you can use to "JSF enable" your Struts apps with minimum fuss—at least in theory! The release we'll review in this lab (version 1.0) has a few significant limitations, which we'll discuss these as the lab progresses.

The approach we'll take is to let you apply the integration library yourself to a very simple "registration" Struts webapp.

■ **Note** The code answers for this lab session are found in the Source Code section of the Apress website, in the file `lab-20-answers.zip`.

Step 1: Preparing the Development Environment

The registration webapp is a pure Struts webapp, which you'll transform so that it uses JSF tags for the View tier instead of the rather limited Struts tags.

1. Unzip the contents of the `registration.zip` file in the Source Code section of the Apress website, to a suitable development folder. This contains the source files for a simple registration webapp.

2. Compile and deploy the registration webapp, then play with it so you know how it works.

Step 2: Install JSF, JSTL, and the Struts-Faces Integration Library

The Struts-Faces integration library requires a JSF 1.0+ implementation and JSTL installed. Unfortunately, the library *does not* work with MyFaces. Instead, you'd have to use Sun's reference implementation, which is not included in the Source Code section of the Apress website due to licensing restrictions. Complete the following:

1. Go to the Sun JSF download site (`http://java.sun.com/j2ee/javaserverfaces/download.html`) and download the latest JSF reference implementation. Extract the relevant JSF binaries, `jsf-api.jar` and `jsf-impl.jar`, and place them in the `.\registration\lib` folder.

2. Extract `jstl.jar` and `standard.jar`, which are the JSTL binaries, from the Struts distribution zip file in the Source Code section of the Apress website to the `.\registration\lib\` folder.

3. From the `struts-faces.zip` file in the Source Code section of the Apress website, extract the JAR file called `struts-faces.jar`, and place it in `.\registration\lib`.

As you should know by now, the scripts for the Registration webapp copy the files in `.\registration\lib` and place them in the webapp's `/WEB-INF/lib` folder upon deployment.

Step 3: Edit web.xml and struts-config.xml

To initialize JSF and get the Struts-Faces library and struts-config.xml files to work, you'll need to follow these steps:

1. Declare the Standard Faces servlet. In `web.xml`, put in the following declaration:

```
<servlet>
     <servlet-name>faces</servlet-name>
     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
     <load-on-startup>1</load-on-startup>
</servlet>
```

   You need to pay special attention to the `<load-on-startup>` value; it has to be the lowest one declared. If you have other servlets declared, then ensure that their `<load-on-startup>` values are greater than the one associated for `FacesServlet`. This ensures that `FacesServlet` gets initialized first.

2. Put in the standard JSF servlet mapping. In `web.xml`, put in the following servlet mapping:

```
<servlet-mapping>
      <servlet-name>faces</servlet-name>
      <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

   This tells Tomcat that the page URLs ending with `.faces` should be shuttled to the servlet called `faces`, which as you can see from the earlier `<serlvet>` declaration is `FacesServlet`.

3. Put in a special Struts Controller. In your `struts-config.xml` file, put in the following `<controller>` element (just *before* the `<message-resource>` tag):

```
<controller>
    <set-property property="processorClass"
     value="org.apache.struts.faces.application.FacesRequestProcessor"/>
</controller>
```

■ **Note** The registration webapp doesn't use Tiles, so we've used the controller shown in the last step. However, if your application *does* use Tiles, use `FacesTilesRequestProcessor` instead.

Step 4: Migrate Your Struts JSP Pages

The Struts-Faces TLD file contains tags that you can use in place of the Struts HTML and Bean tag libraries. You are expected to use JSTL instead of the Struts Logic tag library and nested properties syntax instead of the Nested tag library. The preceding comments apply to the Struts-EL tags as well. You should continue to use the Tiles tag library as it is, though.

If you have an existing Struts application, you can migrate to JSF *incrementally*—you don't have to do it all in one go. If you have to migrate an existing app or you're creating a new one, there are a few things you need to do for each page.

First, declare the JSTL, JSF, and Struts-Faces taglibs in your JSPs. These must be in the following order:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@ taglib prefix="s" uri="http://struts.apache.org/tags-faces" %>
```

Obviously, you should remove declarations to HTML, Bean, Nested, or Logic tag libraries on the page.

Next, put in the replacement tags. Table 20-1 shows the tags in the Struts-Faces taglib, which are equivalents of certain tags in the HTML or Bean taglib.

You could use `<s:loadMessages>` instead of `<f:loadMessages>`. The difference between the two is that `<s:loadMessages>` will make your Struts-declared `Application.properties` file available to JSF tags.

These replacement tags have attributes similar to their Struts counterparts, but in some cases, they are quite rudimentary. A prime example is `<s:message>`, which does not allow replacement arguments, as `<bean:message>` does. In this instance, you should use the much more powerful `<h:outputFormat>` tag, which does allow replacement arguments.

Table 20-1. *Struts-Faces Tags and Their Nearest Pure Struts Equivalents*

Tag Name	Struts Counterpart
s:errors	html:errors
s:form	html:form
s:commandLink	html:link
s:html	html:html
s:write	bean:write
s:message	bean:message
s:javascript	html:javascript

But, in order for JSF tags to read your `Application.properties` file, you'd have to expose it with `<s:loadMessages>` like so:

```
<s:loadMessages var="messages"/>
...
<h:outputFormat value="#{messages['app.logon.prompt.success']} ">
   <f:param value="#{LogonForm.userid}"/>
</h:outputFormat>
```

When there is no equivalent tag, you'd use an appropriate JSF or JSTL tag instead. Table 20-2 contains a few common JSF replacements you can use.

Table 20-2. *Struts Tags and Their JSF Equivalents*

Struts Tag Name	JSF Replacement
`<html:text property="myProp">`	`<h:inputText id="myprop" value="#{myFormBean.myProp}" >`
`<html:password ...>`	`<h:inputSecret ...>`
`<html:hidden ...>`	`<h:inputHidden ...>`
`<html:submit >`	`<h:commandButton id="submit" type="SUBMIT"...>`
`<html:reset >`	`<h:commandButton id="reset" type="RESET"...>`
`<html:cancel >`	`<h:commandButton id="cancel"  type="SUBMIT"...>`

In your migration efforts, you'll have to follow standard JSF rules, like putting in the enclosing `<f:view>` tag. If you're migrating Struts-EL tags, be sure to replace `${...}` with the JSF EL's `#{...}`. The same applies, of course, to the Struts-Faces tags, since they are just JSF extensions.

Now, change the form handler names. In your `<s:form>`s, you need to change the form handler name from `*.do` to the path declared in `struts-config.xml`. For example, if you had

```
<html:form action="MyFormHandler.do" ...
```

this would now be

```
<s:form action="/MyFormHandler" ...
```

Note the leading slash.

Finally, make the necessary changes to `.\registration\web\index.jsp`. You'll probably have to refer to the JSF examples and notes from the previous section.

Step 5: Migrate the <forward>s and Inputs

Lastly, you need to migrate the `<forward>`s that point to the JSPs you've migrated.

■ **Caution** Please pay careful attention here—*only* change a `<forward>` if you've migrated the JSP it points to!

For each such forward, change the `.jsp` suffix of the target page to `.faces`. For example, change

```
<forward name="login" path="/login.jsp" />
```

to

```
<forward name="login" path="/login.faces" />
```

Do *not* amend the actual extensions of the corresponding JSP pages! Similarly, you need to change declarations of all affected input attributes on your `<action-mapping>`s to use the `.faces` extension. For example, you'd change

```
<action .... input="/login.jsp" ...
```

to

```
<action .... input="/login.faces" ...
```

You should only change the inputs that point to migrated pages.

Step 6: Make Entry Points Forward to *.faces

All entry points to your webapp must forward to `*.faces` if that page was migrated to JSF.

For example, suppose the start page for your app was `registration.jsp`, which contained Struts tags. You then migrate this page to JSF. You can't call the page directly (refer to the sidebar "Faces Requests" in the JSF introduction section earlier for details) from the browser.

You need to call it indirectly from another page, say `index.jsp`. This page would contain a forward to `registration.faces`:

```
<jsp:forward page="registration.faces" />
```

Note that there is no leading slash. Make the appropriate change for the Registration webapp.

Step 7: Amend Actions if Necessary

Throughout this book, we've asked you to use `ActionMessages` if you want to report complex validation errors in your `Action` subclasses—and rightly so, since the old `ActionErrors` class has been deprecated.

Unfortunately, the Struts-Faces integration library (version 1.0) only works with this deprecated `ActionErrors` class! If you use the kosher `ActionMessages` class, a `ClassCastException` is thrown when you redisplay a page after a complex validation failure.

I'm sure that this will be remedied in the next version of the library, but in this lab, you're stuck with changing the `ActionMessages` instance in `RegistrationAction` to `ActionErrors`.

Do this now, then compile and deploy your app. You should be able to register a user. Test that all simple and complex validations work. If you can't get everything to work, ensure that you haven't made the following mistakes:

- **Used `#{}` instead of `${}` in your JSTL tags**: JSF tags use the former while JSTL uses the latter.

- **Failed to manually give unique IDs to *every* JSF tag nested within any JSTL tags**: The symptom of this is a "Duplicate ID error" or something similar. In this case, simply give unique IDs to each JSF tag (or JSF extension tag) nested within any JSTL tags. Usually, you can get away with giving the `<s:form>` or `<h:form>` an ID.

If all else fails, don't be ashamed to take a peek at the answers in the `lab-20-answers.zip` file in the Source Code section of the Apress website.

■ **Note** Due to licensing restrictions, the answers zip file does not have the Sun JSF reference implementation. You have to download this yourself.

Step 8: Put in the Necessary <managed-bean> Declarations

This step addresses another shortcoming of the Struts-Faces integration library that you might have noticed if you've got step 7 working: it's that the `RegistrationForm` form bean

does *not* get instantiated when you click the browser's refresh button after successfully registering.

This can be easily fixed by fooling JSF to create the necessary form bean for you. You do this by creating a blank `faces-config.xml` file, then putting in a `<managed-bean>` section with the bean name exactly the same as the one in `struts-config.xml` (in our case, it's `RegistrationForm`).

This is a kludge, and I'm sure the problem will be addressed in future iterations of the Struts-Faces library.

In a Nutshell

We've only scratched the surface of the Struts-Faces integration library in this lab session. If you're interested, you should try out the sample apps that come with the distribution.

You may also have noticed that except for using `ActionErrors` you don't have to amend the `Action` or `ActionForm` subclasses of your app. This of course is a huge plus, and is in most cases much more desirable than migrating to a whole new framework like Shale.

However, you should be aware that the Struts-Faces library won't always work. In an initial draft for this section, I attempted to migrate the login webapp of Chapter 14, only to discover that you can't port Tiles that forward their views to other JSPs, which is what the login webapp does. This is probably a JSF restriction, so it's difficult to see how such issues might be resolved in future versions of the Struts-Faces library. It might be possible to amend the Tiles tags so that they work better with the Struts-Faces integration library.

The lesson here is that you need a new framework that addresses such issues—a reworking of Struts from the ground up. That's what Shale is all about.

Struts Shale Preview

As we've mentioned before, Shale is based on JSF. This means it inherits all of JSF's advantages, like a highly customizable View tier. But Shale adds many benefits on top of JSF. In this section, we'll walk you through the primary value-added areas of Shale. These fall into two broad categories:

- A set of services and service options. (Don't confuse these with web services. The services here are akin to Struts plug-ins, and service options are simply ways to configure Shale and its services.) Services include integration with the Validator framework and a reworking of Tiles, now called "Clay." A significant new addition on top of these is Shale's Dialog Manager, which makes control flow more transparent. The services were designed with customization in mind, so Shale is easier to customize and extend compared with Struts.

- Integration with other frameworks and technologies (Spring, Ajax, JNDI, etc.).

In my opinion, these and the advantages of JSF make Shale a worthy successor of Struts. Unfortunately, technical superiority alone does not guarantee survival, and for the reasons we've outlined in the sidebar at the start of this chapter ("Struts Ti and Struts OverDrive"), Shale's "successorship" from Struts classic is by no means a done deal.

ViewController

In JSF, backing beans are Plain Old Java Objects (POJOs). This gives you a lot of flexibility in creating your backing beans, but it also means that your beans have to do a lot of things "by hand."

A good example is a realistic implementation of the `user` backing bean (see Listing 20-10). In my implementation, I used a `HashMap` to simulate a database. In an actual implementation, you might have to create and release database connections, and these might be best centralized instead of being repeated in functions like `Logon()` and `Register()`.

Shale provides an *interface* called `ViewController` (see Listing 20-13), which contains a few event-handling functions and a variable called `postback`.

Listing 20-13. *Shale's ViewController Interface*

```
/*****************************************************************
Licensed under the Apache License, Version 2.0 (the "License"); you may not use
this file except in compliance with the License. You may obtain a copy of the
License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed
under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and limitations
under the License.
*****************************************************************/

package org.apache.shale.view;

public interface ViewController{

    /**
     * Returns true if the page is called the second time around.
     */
    public boolean isPostBack();
```

```
    /**
     * Sets the "post back" flag.
     */
    public void setPostBack(boolean postBack);


    /**
     * Called after the JSF request processing lifecycle has been
     * completed for the current request.
     */
    public void destroy();


    /**
     * Called after this ViewController backing bean has been
     * instantiated, and after all of the property setters specified
     * above have been called, but before the JSF request processing
     * lifecycle processing has started.
     */
    public void init();


    /**
     * Called after the UI tree has been restored. So, it will NOT be
     * called if the page is displayed for the first time.
     */
    public void preprocess();


    /**
     * Called before the "Render Response" phase. This method
     * will be called only for the view that will actually be
     * rendered. For example, it will not be called if you have
     * performed navigation to a different view.
     */
    public void prerender();
}
```

If your *managed* bean implements the `ViewController` interface, Shale will call the event-handling functions on your `ViewController` implementation at the appropriate times (refer to the comments in Listing 20-13).

The idea is for you to centralize resource creation or destruction activities in these functions.

■ **Note** The class `AbstractViewController` implements `ViewController`, with a functioning implementation for `isPostBack()` and `setPostBack()`, and no-op implementations for the event handlers. If you wish, you can extend this class for convenience.

One important gotcha is that you have to have the right name for your `<managed-bean>`. Not any old name will do! Your managed bean's name (we're referring to the name declared in `faces-config.xml`, not the classname of the backing bean) will have to match the name of the associated the JSP page (referred to as a *view identifier*). Here are some examples:

- If the view identifier is `/logon.jsp`, then the managed bean must be named `logon`.
- If the view identifier is `/admin/logon.jsp`, then the managed bean must be named `admin$logon`.
- If the view identifier is named `/header.jsp`, then the managed bean must be named `_header`, because `header` alone is reserved in JSF.

From this, you can see that there's a one-to-one relationship between managed beans and your JSP pages. This is in contrast to the one-to-many relationship we've used for the registration webapp.

All this might seem a little abstract, so we'll apply the `ViewController` idea to a "realistic" example—the Registration webapp, where database connections need to be created and released. Listing 20-14 shows this class, with the logic stripped out for clarity.

Listing 20-14. *User Backing Bean, Take 2*

```
package net.thinksquared.reg;

//...other import statements omitted

org.apache.shale.view.AbstractViewController;

public class User extends AbstractViewController{

   Connection _connection; //connection to database

   //...other private variables
```

```
    //...constructor

    //...data get/set

    //...UI get/set

    //------------------------------------ Actions

    public String Register(){

       //use _connection to read and write to database

    }

    public String Logon(){

       //use _connection to read and write to database

    }

    //------------------------------------ for ViewController

    public void init(){

       //create _connection to database.

    }

    public void destroy(){

       //release _connection to database.

    }

}
```

You can see in Listing 20-14 how we've taken advantage of the facilities on `ViewController` to centralize creation and release of the database connection.

Of course, because Shale assumes a one-to-one mapping between managed beans and Views, the `<managed-bean>` declaration (see Listing 20-8) has to be changed as shown in Listing 20-15.

Listing 20-15. *New <managed-bean> Declarations for the Registration Webapp*

```
<managed-bean>
  <managed-bean-name>logon</managed-bean-name>
  <managed-bean-class>net.thinksquared.reg.User</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>register</managed-bean-name>
  <managed-bean-class>net.thinksquared.reg.User</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

As you can see, we've used the naming rule we discussed earlier. Also, the `binding` and `value` attributes in `register.jsp` and `logon.jsp` will have to change. Instead of using `#{used.XXX}`, you have to use either `#{logon.XXX}` or `#{register.XXX}`, as appropriate.

Dialog Manager

According to the Shale website, Shale's Dialog Manager is a "mechanism to define a 'conversation' with a user that requires multiple HTTP requests to implement, modeled as a state diagram."

So, a *dialog* is an interaction with the user spanning multiple HTTP requests. Shale lets you model a dialog (you can define more than one dialog) in an XML file, usually called `dialog-config.xml`. You must declare this in your `web.xml` file like so:

```
<context-param>
  <param-name>org.apache.shale.dialog.CONFIGURATION</param-name>
  <param-value>/WEB-INF/dialog-config.xml</param-value>
</context-param>
```

If you have more than one dialog configuration file, simply add them to the declaration using a comma as a separator.

Each "dialog" is modeled as a state diagram. Listing 20-16 shows how we might define a couple of dialogs for the Registration webapp.

Listing 20-16. *Dialogs for the Registration Webapp (dialog-config.xml)*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dialogs PUBLIC
  "-//Apache Software Foundation//DTD Shale Dialog Configuration 1.0//EN"
  "http://struts.apache.org/dtds/shale-dialog-config_1_0.dtd">
```

```
<dialogs>

  <dialog name="Logon" start="Perform Logon">

    <action   name="Perform Logon" method="#{logon.Logon}">
      <transition  outcome="success" target="Exit"/>
      <transition  outcome="register" target="Register User"/>
    </action>

    <subdialog name="Register User" dialogName="Register User">
      <transition  outcome="success" target="Exit"/>
    </subdialog>

    <end  name="Exit" viewId="/success.jsp"/>

  </dialog>

  <dialog name="Register User" start="Registration Form">

    <view name="Registration Form" viewId="/register.jsp">
      <transition outcome="register" target="Perform Registration"/>
    </view>

    <action name="Perform Registration" method="#{register.Register}">
      <transition  outcome="success" target="Exit"/>
    </action>

    <end  name="Exit" viewId="/success.jsp"/>

  </dialog>

</dialogs>
```

The logic of Listing 20-16 should be obvious. Notice that we've used the new `<managed-bean>` declarations of Listing 20-15 here.

■ **Note** The Registration webapp doesn't fully reveal the usefulness of Shale's dialogs, because the user interaction spans only a couple of pages.

Notice that in Listing 20-16 all the transition information is contained in the `dialog-config.xml` file. But in our earlier JSF implementation, the JSP pages invoked the functions on the `user` backing bean directly like so:

```
<h:commandButton action="#{user.Logon}"    ... //see Listing 20-11
<h:commandLink action="register">          ... //see Listing 20-11
<h:commandButton action="#{user.Register}" ... //see Listing 20-12
```

We'll have to change these to

```
<h:commandButton action="dialog:Logon"  ...
<h:commandLink action="dialog:Register User"> ...
<h:commandButton action="register" ...
```

because the navigation is now performed using Shale's Dialog Manager, not by JSF's default navigation handler. Notice that the `action` attributes now begin with `dialog:XXX`, with `XXX` the "name" of the dialog. The command button that submits the registration information (on `register.jsp`):

```
<h:commandButton action="register" ...
```

is different because it needs to return a logical outcome and not enter into a dialog directly:

```
<view name="Registration Form" viewId="/register.jsp">
  <transition outcome="register" target="Perform Registration"/>
</view>
```

Lastly, note that you can mix both JSF-type navigation and Shale's dialog-based navigation. All you need to do in order to enter a Shale dialog is to use a logical outcome named `dialog:XXX`. In our case, we've moved all the navigation elements from Listing 20-8 into the `dialog config` file, so the `faces-config.xml` file should now only contain the `<managed-bean>` declarations.

Exercise

Download the latest copy of the Shale distribution (see "Useful Links") and implement the Registration webapp as we've discussed up to this point.

Integration with the Validator Framework

At the time of this writing, the integration with the Validator framework does *not* include the use of a `validations.xml` file in which you can declare your validations.

Instead, Shale's approach to integration with the Validator framework is very similar to the MyFaces approach of creating extensions to JSF. The one big advantage Shale has is that you can create client-side validations, as you can with Struts.

For server-side validations, however, the MyFaces approach appears cleaner, at least for the moment. Shale is a moving target!

Here's how to use Shale's validator tags:

1. Add `validator-rules.xml` from the Commons Validator distribution to your `WEB-INF` directory. You can use the one that comes with classic Struts.

2. Put in this taglib declaration: `<%@ taglib uri="http://struts.apache.org/shale/core" prefix="s" %>`.

3. Add Shale's validators to JSF input components with `<s:commonsValidator>`.

If you want client-side validation, you need to add the extra attribute `onsubmit='validateForm(this)'` and the child tag `<s:validatorScript functionName="validateForm"/>` to the `<h:form>`. The former invokes the JavaScript functions to validate the form, while the latter instructs Shale to paste in the JavaScript validation code.

Listing 20-17 shows how we might rewrite `logon.jsp` (see Listing 20-11) using Shale's Validator framework. As usual, we've stripped out the `<f:verbatim>` tags.

Listing 20-17. *logon.jsp Using Shale*

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://struts.apache.org/shale/core" prefix="s"%>
<f:loadBundle basename="net.thinksquared.reg.messages"
                   var="reg_messages"/>
<html>
<body>
  <f:view>
    <h:panelGroup>
      <h:messages style="color:red" globalOnly="true"/>
      <h:outputText value="#{reg_messages['title_reg']}"
                    style="font-weight:bold" />

      <h:form binding="#{user.container}"
             onsubmit="validateForm(this)">

        <h:outputText value="#{reg_messages['user_id']}"/>
```

```
        <h:inputText id="userId"
                  value="#{logon.userId}" required="true">
          <s:commonsValidator type="mask"
                              mask="^[A-Za-z0-9]{5,10}$"
                               arg="#{reg_messages['bad_user_id']}"
                            server="true"
                            client="true"/>
        </h:inputText>

        <h:message for="userId" />

        <h:outputText value="#{reg_messages['password']}"/>

        <h:inputText id="password" value="#{logon.password}"
               required="true">
          <s:commonsValidator type="mask"
                              mask="^[A-Za-z0-9\-\_]{5,10}$"
                               arg="#{reg_messages['bad_user_id']}"
                            server="true"
                            client="true"/>
        </h:inputText>

        <h:message for="password" />

        <h:commandButton action="dialog:Logon"
                         value="#{reg_messages['logon']}"/>

        <s:validatorScript functionName="validateForm"/>

      </h:form>

      <h:commandLink action="dialog:Register User">
        <h:outputText value="#{reg_messages['new_user']}" />
      </h:commandLink>

    </h:panelGroup>
  </f:view>
</body>
</html>
```

Notice that we've put in the changes we made in the previous two subsections.

JNDI Integration

JNDI (Java Naming and Directory Interface) is a Java Enterprise Edition feature that obtains handles to resources that are declared in either `web.xml` or configuration settings of an application server.

A thorough discussion of JNDI is outside the scope of this book. However, the Shale website gives a couple of simple examples, variations of which we'll describe here.

JNDI addresses the concept of *environment entries*, which are simply variable declarations in `web.xml`. For example, suppose your webapp had to display your company's logo on each page. You could use a message resource to store the URL, but another way is to use an environment entry (I'm not recommending this method, but just using it as an illustration).

First, you declare the environment entry in `web.xml`:

```
<env-entry>
  <description>URL to company logo GIF</description>
  <env-entry-name>logo</env-entry-name>
  <env-entry-value>http://www.myco.myapp/logo.gif</env-entry-value>
  <env-entry-type>java.lang.String<env-entry-type>
</env-entry>
```

Your JSF tags can use Shale's JNDI integration feature to access this environment entry:

```
<h:graphicImage value=#{jndi:logo} />
```

That's it. The `jndi:` prefix tells Shale that the binding refers to a JNDI resource, in this case, the one environment entry named `logo`.

A more realistic scenario involves getting a database connection. JNDI allows you to declare *data source references* in `web.xml`. For example, you might define a data source reference to your company's personnel database like so:

```
<resource-ref>
  <description>MyCo Personnel Database</description>
  <res-ref-name>jdbc/MyCoPersonnelDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Typically, you'd want to create and release the database connection in your backing beans. Listing 20-18 shows how we can use Shale's JNDI integration to provide an implementation of `init()` of Listing 20-14.

Listing 20-18. *An Implementation of User.init()*

```
public void init(){
 //get faces context
 FacesContext context = FacesContext.getCurrentInstance();

 //create _connection to database.
 ValueBinding vb =
   context.getApplication()
   .createValueBinding("#{jndi['jdbc/MyCoPersonnelDB'].connection}");

 _connection = (Connection) vb.getValue(context);

}
```

Reusable Views with Clay

Clay is a subframework of Shale. It is essentially a custom JSF custom UI component, which acts as a stand-in for a sub-UI tree defined elsewhere. This idea should not be new to you. Tiles does something similar—you define the UI in a `tiles-def.xml` file, then use a Tiles tag to indicate where you want the UI placed.

Clay does the same, but provides more options for you to build the UI tree. Any meaningful exploration of Clay would require a whole chapter by itself, so I will leave it to you to explore this interesting technology on your own.

Server-Side Ajax Support

Ajax is not a single technology but refers to a broad spectrum of technologies characterized by their use of client-side JavaScripts and XML to pass data between client and server.

Server-side Ajax support (aka *remoting*) involves extracting data from the Model tier and serializing it as XML for delivery to the client.

Shale's remoting support is mainly focused on the delivery of XML data to the client (e.g., the `org.apache.shale.remote.ResponseWrapper` class to write XML), and management of the remote sessions.

Test Framework

Unit testing is the process of incrementally building and testing your code. In the unit-testing paradigm, you code a little, then test a little, and continue until your app is done. The tests you create are Java classes that you need to run in sequence, each testing portions of your code. JUnit (see "Useful Links") is *the* universally used unit-testing framework of the Java world.

Believe me, unit testing can save you a lot of trouble, even on moderately complex applications. Once you've tried it, you'll wonder how you ever managed to produce Java code in the past.

Unfortunately, unit testing has made little inroads into the testing of webapps. Shale supports unit testing by providing you with a set of mock objects and JUnit test case base classes. Check out the JUnit website for more details.

Exercise

Download the latest Shale distribution (see "Useful Links") and try out the example `use-cases` webapp.

JSF vs. Shale vs. Struts

Ultimately, which technology you choose (JSF, Shale, or Struts) depends on your requirements. Having said that, if you're comfortable with the Apache offerings, then you're likely to commit to Shale over pure JSF. Shale brings so much to the table that using it over pure JSF is a no-brainer.

The Achilles' heel of Shale is that it's in a state of flux, like any new piece of software. At the time of this writing many of the services are marked as "Developing," meaning they could change at any time. I certainly expect Shale to change substantially by the time you read this book. So, if you're planning to use Shale, you should carefully research the features you need to ensure that they are fairly stable. The Struts mailing list (Shale shares this with classic Struts, so you should mark your emails with [Shale]) and the Shale documentation are good resources.

The choice between Struts and Shale is more difficult to make. If you have an existing Struts webapp, it's advisable to continue using Struts, because the cost of porting your existing JSP pages to using JSF tags is usually prohibitive.

For new projects, it's difficult to give a clear answer. The Struts developers themselves and those involved in JSF and Shale appear to unconditionally advocate moving to Shale or JSF. I would be more circumspect, because as you've seen, JSF (and therefore Shale) is more complex than Struts. It isn't rocket science, but there's definitely a sharp learning curve involved.

However, if you find you've been using strange tricks to force Struts into getting your webapp to behave like a desktop application, then Shale is definitely something you'd want to explore along with other web application frameworks like Spring.

I've avoided giving you a feature-by-feature comparison of Struts, JSF, and Shale. Although they are all, broadly speaking, web application frameworks, their "sweetspots" (the scenarios in which they shine) are different.

Instead, Table 20-3 compares classic Struts, JSF, and Shale to help you answer a simple question: "I'm starting a new project. Should I use Struts or Shale or JSF?" Take the ratings as being *relative* to each other. Please don't take them to be absolute in any sense—"easy" for example, means "easy compared to" the other options.

Table 20-3 doesn't take into account the Struts-Faces integration library. Even if it did, probably not much would change. Struts suffers from design issues that won't go away easily. These deficiencies (poor extensibility, code modularity and code reuse) come to the fore when you're developing more complex applications. The Struts-Faces library, though useful, doesn't address architectural issues.

Table 20-3. *Comparison Matrix for Struts, JSF and Shale*

Factor	Struts	JSF (MyFaces)	Shale	Comments
Learning curve	Gentle	Average	Steep	Shale's learning curve is steepest because it introduces additional services on top of JSF.
Developing simple webapps	Easy	Difficult	Average	By a "simple" webapp I mean something like LILLDEP in complexity. Many webapps currently fall into this category. Shale, with its Dialog Manager, improves page transition maintenance. This is a definite plus over pure JSF.
Developing complex webapps	Very difficult	Average	Average/Easy	By "complex" webapps I mean a webapp that (for example) replicates the full functionality of OpenOffice Writer (or Word). Shale, with its extensibility, and extras (like support for Ajax, dialogs, etc.) makes life much better.
Extensibility	Poor	Very good	Good	Struts only offers plug-ins for extensibility (and more limited extensibility in the form of custom loaders for the configuration file). You can do a lot with the extensibility points that Struts offers, but you'd have to reinvent the wheel sometimes, as I did in Chapter 19. Both JSF and Shale offer much more structured ways in which you can extend them.
Longevity	High	Very high	Moderate/Low	By longevity, I mean "What's the probability of this framework being around in five years' time?" These are my forecasts: JSF's survival is assured, given Sun's endorsement of it as a spec, and as the only credible components-based web framework for the Java platform. Struts classic will probably survive because of its huge user base. Shale might be overshadowed by competing technologies in the same niche, ones that don't pretend to be a "Struts successor." So it's very unfortunate that it's named *Struts* Shale. People then tend to think in terms of Struts classic versus Struts Shale, when in fact, they best apply to different niches.

Table 20-3. *Comparison Matrix for Struts, JSF and Shale*

Factor	Struts	JSF (MyFaces)	Shale	Comments
Ease of upgrading	Easy	Easy	Difficult	If classic Struts remains backward compatible, then obviously it's easy to upgrade. JSF is a mature spec, so you shouldn't expect surprises there either. Shale, on the other hand, is changing (at the time of this writing), so you might experience difficulty upgrading to a newer version.
Code reuse	Low	High	Average	Code reuse becomes important in more complex projects, so the ratings given here apply mainly to them. JSF was built with reuse and extensibility foremost.
Modular Code	Low	Average	Average	It's difficult to make modular code with Struts for complex webapps. You might have gotten a glimpse of this in Lab 14. It's easy to create modular code with JSF if you know how, but it's also easy to make a mess of it. Code modularity impacts maintenance and development. The more modular the code, the easier it is to maintain and develop. See the answer for the question raised in Chapter 1, given in Appendix D, for a short discussion of this.

Useful Links

- The official JSF specification: `http://java.sun.com/j2ee/javaserverfaces/download.html`
- MyFaces website: `http://myfaces.apache.org/`
- Struts-Faces integration library download site: `http://jakarta.apache.org/site/binindex.cgi`
- The Sun JSF download site: `http://java.sun.com/j2ee/javaserverfaces/download.html`
- The official Shale website: `http://struts.apache.org/struts-shale/index.html`
- Struts Ti website: `http://struts.apache.org/struts-sandbox/struts-ti/index.html`
- WebWork website: `www.opensymphony.com/webwork/`
- Struts OverDrive website: `http://opensource2.atlassian.com/confluence/oss/display/OVR/Home`

- Spring website: `www.springframework.org`
- *Foundations of Ajax* (Apress, 2005): `www.apress.com/book/bookDisplay.html?bID=10042`
- CSS Zen Garden: `www.csszengarden.com`
- Subversion website: `http://subversion.tigris.org` (and see `http://svnbook.red-bean.com` for *Version Control with Subversion*, a free book about Subversion)
- Ant website: `http://ant.apache.org`
- JUnit page: `www.junit.org`
- The JSF Central website: `www.jsfcentral.com`
- The JSF FAQ website: `www.jsf-faq.com`

Summary

- JSF is the new Java standard for the View tier of modern Java webapps.
- JSF is a specification. One open source implementation is MyFaces from Apache.
- The primary purpose of JSF is the management of UI components as displayed in a web browser.
- JSF goes a long way in allowing the creation of webapps that behave like desktop applications.
- Shale extends JSF past the View tier to include integration with Apache Commons projects (like Validator), improves complex navigation (using dialogs), and much more.
- Despite its name, Struts Shale is not a "drop-in" replacement for classic Struts.
- Struts and JSF/Shale serve different niches. Struts works well for simple webapps, but not for more complex ones. Shale addresses the needs of more complicated web applications.
- If you're interested in using JSF *incrementally* with Struts without having to completely redesign your webapp, consider using the Struts-Faces integration library.

APPENDIX A

Frameworks for the Model

In this appendix, I'll provide simple examples of how to use Hibernate and Torque, and also (shameless plug) a simple, open source, "classes that persist" framework called Lisptorq, developed by the company I work for.

I'll illustrate the use of these persistence frameworks with the Registration webapp I first introduced in Chapter 5. I'll show you how to persist different versions of the `User` class outlined in Listing A-1.

Listing A-1. *Skeleton for the Registration Webapp Model*

```
public class User{

   protected String _userId = null;
   protected String _password = null;

   /* Bean-like getters and setters */
   public String getUserId(){
      return _userId;
   }

   public String setUserId(String userId){
      _userId = userId;
   }

   public String getPassword(){
      return _password;
   }

   public String setPassword(String password){
      _password = password;
   }
```

```
    /**
    * Checks if the userid exists.
    */
    public static boolean exists(String userid){
        //implementation omitted
    }

    /**
    * Saves the _userid and _password pair
    */
    public void save() throws Exception{
        //implementation omitted
    }
}
```

Getting the Software

All three persistence frameworks discussed here are open source, and may be obtained freely from their websites:

- **Hibernate**: `www.hibernate.org`
- **Torque**: `http://db.apache.org/torque/`
- **Lisptorq**: `www.thinksquared.net/dev/lisptorq/`

Both Hibernate and Torque require some setup in order for you to use them in your projects. Consult their respective websites for installation instructions.

Lisptorq, on the other hand, was specially designed to be easy to use and experiment with. You have to copy only two JAR files: a Java Lisp interpreter called jlisp and the Lisptorq JAR file, `lisptorq.jar`, which contains the LISP programs that constitute Lisptorq. Both of these are in the Source Code section of the Apress website, found at `http://www.apress.com`, but you should download the latest files from the URL listed earlier.

Because Lisptorq is simple to understand and use, I'll introduce it first, and give you some practice using it in Lab A. I'll then redo the Registration webapp using Hibernate and Torque.

Lisptorq

Lisptorq is a *code generator,* and it produces Java source code given a database schema. These generated classes constitute the Model tier of your webapp.

As an example, Listing A-2 shows how you might define the database schema for the Registration webapp using Lisptorq. There's just one table called `user` with two fields, `user_id` and `password`.

Listing A-2. *Defining the user Table in Lisptorq*

```
define database
(
   (user
       (user_id    (REQUIRED PRIMARY (VARCHAR 32)))
       (password   (REQUIRED (VARCHAR 32)))
   )
)
```

Listing A-2 should be self-explanatory. Both the `user_id` and `password` fields are mandatory, and the `user_id` field is used as a primary key.

When Lisptorq is run on the schema described by Listing A-2 (I'll show you how in the following lab), the output is *four* Model classes, plus four other helper classes. In general, for each table on the database schema, four Model classes are generated.

The first two are called *data object* classes, which are really JavaBeans with properties for each column on the table. Persistence is handled by *peer classes*, which have methods for saving, deleting, and searching for data objects on the database. So, the generated Model classes for Listing A-2 using Lisptorq would be

- `BaseUser`: Contains getters and setters for `user_id` and `password`. It also contains a `save()` method to persist the data in the class.
- `BaseUserPeer`: Contains functions to persist a `BaseUser` class. It contains functions like `doSave()`, `doDelete()`, and `doSelect()`, which save, delete, and search for `Users`.
- `User`: An empty class representing the user. You can write your own additional functions in this class. `User` extends `BaseUser`.
- `UserPeer`: An empty subclass of `BaseUserPeer`. This class lets you extend the functionality of `BaseUserPeer`.

Having four classes for one database entity might seem a little complicated, but there's a method to the madness. The `User` class (a data object) holds data while the `UserPeer` classes (peer classes) are used to save, delete, or retrieve data objects from the database. The `Base` classes contain autogenerated functions that do the real work. The non-`Base` classes (`User` and `UserPeer`) are given to you to add additional functionality on top of the `Base` classes.

Listing A-3 shows how you might save the user ID and password data using the `User` class.

Listing A-3. *Saving a Lisptorq-Generated User Bean*

```
User u = new User();
u.setUserId("kolmogorov");
u.setPassword("turbulence23");
u.save();
```

These few lines do a lot behind the scenes: SQL is generated to save the user ID and password. Using Model classes saves you from writing a single line of SQL yourself!

In addition to the four model classes (remember, four are created per table), Lisptorq always generates four helper classes: `Criteria`, `Database`, `DatabaseSetup`, and `Scroller`:

- `Criteria` allows you to create queries without using SQL.
- `Database` manages connections to the database.
- `DatabaseSetup` gives you a `main()` function you can call to set up your database. You'd obviously need to use this class just once.
- `Scroller` is an *interface* that extends `java.util.Iterator`. It represents an iterator that can iterate either forward (using the usual `hasNext()` and `next()`) or backward (using `hasPrevious()` and `previous()`). You may also position the "cursor" using `absolute()`.

Listing A-4 is an implementation of the `exists()` function of Listing A-1 using the `Criteria` class as well as the `User` and `UserPeer` Model classes.

Listing A-4. *An Implementation of exists() using the Criteria and Model Classes*

```
public static boolean exists(String userId){

  Criteria crit = new Criteria();
  crit.add(User.USER_ID,userId);
  return UserPeer.doSelect(crit).iterator().hasNext();

}
```

Lisptorq currently produces generated code optimized for Derby SQL. Derby is a Java embedded database, currently being "incubated" at Apache. It was donated by IBM, and was previously the proprietary Java embedded database called Cloudscape. It is one of the few open source databases available that support Atomicity, Consistency, Isolation, and Durability, or ACID (See "Useful Links"). The JAR file for Derby (`derby.jar`) is in the Source Code section of the Apress website at `http://www.apress.com`.

Lab A: Test Driving Lisptorq

I hope the previous section has piqued your interest in using Model classes. To get your toes wet, in this lab session you'll develop a simple application to store and retrieve data using Lisptorq-generated Model classes.

In order for you to complete this lab, you must refer to the Lisptorq manual at `www.thinksquared.net/dev/lisptorq/`.

Step 1: Preparing the Development Environment

1. Create a folder in your development environment called `lisptorq`.
2. Create a `.\lib` subfolder within this folder.
3. Extract `jlisp.jar`, the LISP interpreter found in `jlisp_0.2.zip` in the Source Code section of the Apress website at `http://www.apress.com`, into `.\lib`.
4. Save `lisptorq.jar` in the Source Code section of the Apress website into `.\lib`.
5. Save `derby.jar`, the Derby database driver in the Source Code section of the Apress website, into the development `.\lib` folder.

Step 2: Writing the Database Schema

In what follows, we will create a simple bookstore app, in which we'll store books along with publisher and author information. (This app originally appeared in Apache Torque's tutorial.) The database has three tables:

- `author`: Has four columns: `author_id`, `first_name`, `last_name`, and `full_name`. Make `author_id` an autogenerated field. It should also be a primary key for this table.
- `publisher`: Has just two columns: `publisher_id` and `name`. Make `publisher_id` an autogenerated field. It should also be a primary key for this table.
- `book`: Has four columns: `title` and `ISBN` to hold data; `author_id`, which is a foreign key to the `author_id` column in the `author` table; and `publisher_id`, which is a foreign key to the `publisher` table.

It should be fairly transparent what the three tables represent. Apart from the three ID columns, which should be `INTEGER`s, the other columns should be `(VARCHAR 255)`.

You need to complete the following:

1. Create a new text file called `main.lisp` in the Lisptorq development folder.

2. Open this up with Notepad (or your favorite IDE), and write the Lisptorq database schema for this database. You will certainly need help—please refer to the Lisptorq manual at `www.thinksquared.net/dev/lisptorq/`.

Step 3: Specifying the Database Settings

Derby is an embedded database, so you have to give it a folder to store the database files, and you have to specify a system property called `derby-home`. These tasks are done for you by Lisptorq's generated helper classes, but you have to supply the information.

Here are the configuration parameters you must specify:

- `database-type` should be set to `derby`.
- `database-name` should be set to `bookstore`.
- `derby-home` should be set to `./bookstore`.
- `package` should be set to `net.bookstore`.

Again, refer to the online manual for help on how to set these configuration parameters.

`database-type` is set to `derby` because we're using the Derby database for this lab. `database-name` specifies how the generated classes will internally refer to the database. The `derby-home` setting is required by Derby, and is a path that points to the folder that contains the database files. You do *not* have to create this folder yourself; it will be done for you by the `DatabaseSetup` program. Finally, the `package` setting is the package in which you would like to place the generated classes.

Step 4: Generate the Java Files

First, enter (`lisptorq:generate database`) at the end of `main.lisp`. This command is a request to generate the Model classes. Next, run `main.lisp` in order to generate all the Model tier source code files:

```
java -classpath .\lib\jlisp.jar;.\lib\lisptorq.jar
   net.thinksquared.jlisp.Loader main.lisp
```

If all goes well, you should see 16 files (four for each table plus four helper classes) in the development folder.

The next step is to write a test program to write and retrieve records.

Step 5: Writing the Test Program

Write a simple test program in the file `Test.java` to store the following information:

- `Authors`: Lewis Carroll, Harold Edwards, Arthur Conan Doyle
- `Publishers`: Dover Books, Macmillan Publishers
- `Books`:
  - *Alice in Wonderland*, by Lewis Carroll, Dover Books
  - *Riemann's Zeta Function*, by Harold Edwards, Dover Books
  - *The Sherlock Holmes Casebook*, by A. C. Doyle, Macmillan Publishers
  - *Through the Looking Glass*, by Lewis Carroll, Macmillan Publishers

Enter this information into the relevant data objects, and then save them. Remember that `Books` require foreign keys. How will you specify these?

Once the data is saved, run the following queries:

- List all books and their authors from Dover Books.
- List all the publishers that carry books by Lewis Carroll.

Again, you should refer to the online manual for help.

Step 6: Initializing the Database

Compile your work (`javac *.java`) and then run `DatabaseSetup` to create the database:

```
java  -classpath ./lib/derby.jar org.bookstore.DatabaseSetup
```

You should see the database files created in the `.\bookstore` folder.

Step 7: Running the Test Program

Finally, run your Test program with the command

```
java -classpath ./lib/derby.jar org.bookstore.Test
```

Play with this simple application, until you are thoroughly familiar with the Model classes. Try to extend the functionality beyond that described here. Examine the database schema for LILLDEP (described in a later section in this appendix).

Next, let's see how to use Torque for the Registration webapp.

Using Torque for the Registration Webapp

Like Lisptorq, Torque also autogenerates Java source code for you. In fact, the data object and peer classes and the functions on them are very similar to those generated by Lisptorq (that is, in fact, where the *torq* in *Lisptorq* comes from).

I won't discuss the installation and setup of Torque here—you should refer to the Torque website (see "Useful Links") for details. However, I will illustrate its use with the Registration webapp. Unlike Lisptorq, Torque uses XML to specify the database schema. The `user` table is described in Torque's XML schema in Listing A-5.

Listing A-5. *Torque's XML Description of the user Table*

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE database SYSTEM
 "http://db.apache.org/torque/dtd/database_3_1.dtd">

<database name="registration">

  <table name="user" description="Table of registered users">

    <column
      name="user_id"
      required="true"
      primaryKey="true"
      type="VARCHAR"
      size="32"/>

    <column
      name="password"
      required="true"
      type="VARCHAR"
      size="32"/>

  </table>

</database>
```

The XML is quite self-explanatory. Torque's output from this XML consists of the usual four classes (data object and peer classes), as it did for Lisptorq. So if you know how to use Lisptorq, you know how to use Torque.

For example, the `exists()` function (see Listing A-4) using Torque is provided in Listing A-6. As you can see, compared with Listing A-4 there is very little difference.

Listing A-6. *An Implementation of exists() Using Torque*

```
public static boolean exists(String userId){

  Criteria crit = new Criteria();
  crit.add(UserPeer.USER_ID,userId);
  return UserPeer.doSelect(crit).iterator().hasNext();

}
```

The central idea behind both these frameworks is that you have special Model classes, custom built by the framework for your webapp. Hibernate takes a different tack, allowing you to persist *any* JavaBean. Unfortunately, with flexibility comes more complexity, but Hibernate does an admirable job of keeping this to a minimum. We'll see how Hibernate works for the Registration webapp next.

Using Hibernate for the Registration Webapp

Hibernate can persist any Java object or collection of objects, as long as they follow the JavaBeans convention: each property of the Java class must have associated `getXXX` and `setXXX` methods.

As with Torque, some setup is required in order for you to use Hibernate. Consult the Hibernate website (see "Useful Links") for details. In this section, I'll focus on showing how you might use Hibernate as the Model tier for the Registration webapp.

The `User` class (reproduced in Listing A-7) does follow this convention, since both the `_userId` and `_password` variables have associated getters and setters.

Listing A-7. *The User Bean*

```
package net.thinksquared.registration.data;
public class User{

   protected String _userId = null;
   protected String _password = null;

   public String getUserId(){
      return _userId;
   }

   public String setUserId(String userId){
      _userId = userId;
   }
```

```
    public String getPassword(){
        return _password;
    }

    public String setPassword(String password){
        _password = password;
    }

}
```

For each bean you want to persist using Hibernate, you have to create an XML description of it, which essentially maps the bean's properties to tables and columns in your database. The XML mapping for `User` appears in Listing A-8.

Listing A-8. *Hibernate XML Description of the User Bean*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
    <class name="net.thinksquared.registration.data.User"
           table="user">

        <id name="userId" type="string"
            unsaved-value="null">
            <column name="user-id" sql-type="char(32)"
                    not-null="true"/>
            <generator class="assigned"/>
        </id>

        <property name="password">
            <column name="password" sql-type="char(32)"
                    not-null="true"/>
        </property>

    </class>
</hibernate-mapping>
```

The XML is fairly self-explanatory, and I won't describe it further in any detail. The essential idea is that you have an XML file that maps your bean to an existing table in the database.

Saving a bean from scratch is a simple three-step process, very similar to saving data with JDBC:

1. Initialize Hibernate.
2. Obtain a `Session` object.
3. Save the bean using the `Session` and close the `Session`.

Step 1 is usually performed *once*, when the application starts up. Listing A-9 illustrates steps 1 to 3.

Listing A-9. *Saving a User Bean with Hibernate*

```
User u = new User();
u.setUserId("kolmogorov");
u.setPassword("turbulence23");

//Step 0: Start Hibernate
Configuration config = new Configuration().addClass(User.class);
SessionFactory factory = config.buildSessionFactory();

//Step 1: Get a Session
Session s = factory.openSession();

//Step 2: Save and close
Transaction t = s.beginTransaction();
s.save(u);
t.commit();
s.close();
```

Remember that the `User` class is not autogenerated by Hibernate—it's assumed to be an existing class in your app. Hibernate's job is to allow you to save instances of `User` to a database without you having to worry about SQL and databases.

Doing a search for an object is easy, too. Listing A-10 shows how to implement the `exists()` function to check whether a given user ID exists.

Listing A-10. *An Implementation of exists() with Hibernate*

```
public static boolean exists(String userId){

  final String QUERY = "select u from u " +
                       "in class net.thinksquared.registration.data" +
                       " where u.userId = :userId";

  Session s = ... //Get a session somehow.
  Query q = s.createQuery(QUERY);

  try{

    q.setString("userId",userId);
    List l = q.list();
    return (l.size() > 0);

  }finally{
    s.close();
  }

}
```

The `QUERY` string is an example of the Hibernate Query Language (HQL), which is similar to SQL except that it works for persisted objects.

These examples do not even begin to scratch the surface of Hibernate's capabilities. But I hope I've given you at least some idea of how Hibernate is used.

In a Nutshell...

In this section I've provided a very brief overview of the major persistence frameworks, Hibernate and Torque. These frameworks are complex and whole books are devoted to them, so the treatment in this section is very much incomplete. Even so, I hope I've given you a good starting point for your own explorations into this area.

Autogenerating LILLDEP Model Classes

I've used Lisptorq to generate LILLDEP Model classes. The code for this is in the `.\lisp\` subdirectory of the main LILLDEP development folder.

There are three database tables:

- CONTACT, which stores contact information (see Lab 6)
- COLLECTION, which stores a collection's name and notes
- COLLECTION_MAP, which stores the actual contacts that go into a collection

To use Lisptorq to generate these classes, you first create a LISP file describing the three tables, as in Listing A-11.

Listing A-11. *models.lisp, the Lisptorq LISP File to Autogenerate LILLDEP Model Classes*

```
(import lisptorq.lisp)

define database
(
    (contact
        (contact_id   (PRIMARY REQUIRED INTEGER AUTO))
        (name ((VARCHAR 1024)))
        (designation ((VARCHAR 1024)))
        (department ((VARCHAR 1024)))
        (email ((VARCHAR 255)))
        (tel ((VARCHAR 255)))
        (fax ((VARCHAR 255)))
        (company ((VARCHAR 1024)))
        (address ((VARCHAR 1024)))
        (postcode ((VARCHAR 16)))
        (website ((VARCHAR 255)))
        (activity ((VARCHAR 2048)))
        (classification ((VARCHAR 16)))
        (memo ((VARCHAR 2048)))
    )

    (collection
        (collection_id   (PRIMARY REQUIRED INTEGER AUTO))
        (name ((VARCHAR 1024)))
        (memo ((VARCHAR 2048)))
    )

    (collection_map
        (collection_id   (REQUIRED INTEGER (FOREIGN collection)))
        (contact_id      (REQUIRED INTEGER (FOREIGN contact)))
    )
)
```

```
(lisptorq:set database-name   lilldep)
(lisptorq:set database-type   derby)
(lisptorq:set derby-home    ./dbase)
(lisptorq:set package       net.thinksquared.lilldep.database)


(lisptorq:generate database)
```

To autogenerate the Model classes from this definition, simply put the JAR files `jlisp.jar` and `lisptorq.jar` in your classpath, and call the `Loader` class, like so:

```
java -classpath jlisp.jar;lisptorq.jar
            net.thinksquared.jlisp.Loader models.lisp
```

This will generate all your Model and helper classes, including a `DatabaseSetup` class, which you can call to create the tables on the database. For a tutorial on using Lisptorq, see `www.thinksquared.net/dev/lisptorq/`.

Unfortunately, as for LILLDEP, we'll have to use two different `derby-homes`: one when the database is initialized for the first time, and the other when it is deployed on Tomcat. This means that you've got to create the class twice, or *manually* change the static section that sets up `derby-home` in `Database.java` to that listed in Listing A-12.

Listing A-12. *A Tweak to Database.java*

```
//set path to database
String catalinaHome = System.getProperty("catalina.home");
if(catalinaHome == null){
    //we are called from an pre-install program
    System.setProperty("derby.system.home","./dbase");
}else{
    //we are on a Tomcat servlet
    System.setProperty("derby.system.home",
                        catalinaHome + "/webapps/lilldep/dbase");
}
```

Useful Links

- The Derby project: `http://incubator.apache.org/derby/`
- For more on ACID: `http://en.wikipedia.org/wiki/ACID`
- Hibernate: `www.hibernate.org/`
- Torque: `http://db.apache.org/torque/`
- Lisptorq: `www.thinksquared.net/dev/lisptorq/`

APPENDIX B

Commonly Used Classes

This appendix describes selected servlet and Struts classes. Please note that *not* all functions on a class are shown—only the ones I believe you might find useful.

The documentation for the Struts classes (`Action`, `ActionForm`, `ActionMessage`, `ActionMessages`, `ActionError`, `ActionErrors`, `ActionMapping`, `FormFile`, `ComponentContext`, and `ExceptionHandler`) has been adapted from Apache's JavaDocs for those classes, and is reproduced here for your convenience under the Apache License. A copy of the Apache License is available at `www.apache.org/licenses/LICENSE-2.0`.

javax.servlet.http.HttpServletRequest

This class encapsulates an incoming HTTP request. It is used to store or retrieve objects that have request scope (using `get/set/removeAttribute()`). You may also store session scope objects on it by first using `getSession()` to acquire an `HttpSession` object (see the next section).

Return Type	Function	Comments
`Object`	`getAttribute(String name)`	Returns the value of the named attribute as an `Object`, or `null` if no attribute of the given name exists
`void`	`removeAttribute(String name)`	Removes an attribute from this request
`void`	`setAttribute(String name,Object o)`	Stores an attribute in this request
`HttpSession`	`getSession()`	Returns the current session associated with this request, or if the request does not have a session, creates one
`String`	`getParameter(String name)`	Returns the value of a request parameter as a `String`, or `null` if the parameter does not exist

javax.servlet.http.HttpSession

As its name implies, `HttpSession` is used to store session-scoped variables. It has `get/set/removeAttribute()` functions, identical to `HttpServletRequest`.

Return Type	Function	Comments
`Object`	`getAttribute(String name)`	Returns the object bound with the specified name in this session, or `null` if no object is bound under the name
`void`	`removeAttribute(String name)`	Removes the object bound with the specified name from this session
`void`	`setAttribute(String name, Object value)`	Binds an object to this session, using the name specified
`String`	`getId()`	Returns a string containing the unique identifier assigned to this session

org.apache.struts.action.ActionMessage

`ActionMessage` encapsulates a single locale-independent *message*. Only the constructors of `ActionMessage` are important to webapp developers.

Constructor	Comments
`ActionMessage(String key)`	Constructs an action message with no replacement values
`ActionMessage(String key, Object value0)`	Constructs an action message with the specified replacement values
`ActionMessage(String key, Object[] values)`	Constructs an action message with the specified replacement values
`ActionMessage(String key, Object value0, Object value1)`	Constructs an action message with the specified replacement values
`ActionMessage(String key, Object value0, Object value1, Object value2)`	Constructs an action message with the specified replacement values
`ActionMessage(String key, Object value0, Object value1, Object value2, Object value3)`	Constructs an action message with the specified replacement values

org.apache.struts.action.ActionMessages

This class is used to store multiple ActionMessages.

Return Type	Function	Comments
void	add(String property,ActionMessage message)	Adds a message to the set of messages for the specified property
boolean	isEmpty()	Returns true if there are no messages recorded in this collection, or false otherwise

ActionMessages also has a static String variable called GLOBAL_MESSAGE, which is a property name marker to use for global messages (as opposed to those related to a specific property).

org.apache.struts.action.ActionErrors

ActionErrors is a subclass of ActionMessages, and its sole use is as the return value of ActionForm's validate(). All of the functions and properties on this class have been deprecated. You should refer to functions or properties on ActionMessages instead.

org.apache.struts.action.ActionMapping

ActionMapping holds information about the valid "next" pages for a particular form handler.

Return Type	Function	Comments
ActionForward	findForward(String name)	Finds and returns the ActionForward instance defining how forwarding to the specified logical name should be handled
ActionForward	getInputForward()	Gets the context-relative path of the input form to which control should be returned if a validation error is encountered

org.apache.struts.action.Action

An Action is an adapter between the contents of an incoming HTTP request and the corresponding business logic that should be executed to process this request. Struts will select an appropriate Action for each request, create an instance (if necessary), and call the

execute() function. Actions must be programmed in a thread-safe manner, because Struts will share the same instance for multiple simultaneous requests. This means you should design with the following points in mind:

- Instance and static variables *must not* be used to store information related to the state of a particular request.
- They *may* be used to share global resources across requests for the same Action.
- Access to other resources (JavaBeans, session variables, etc.) *must* be synchronized if those resources require protection. (Generally, however, resource classes should be designed to provide their own protection where necessary.)

Return Type	Function	Comments
ActionForward	execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)	Processes the specified HTTP request, and creates the corresponding HTTP response (or forward to another web component that will create it), with provision for handling exceptions thrown by the business logic
void	addErrors(HttpServletRequest request, ActionMessages errors)	Adds the specified errors keys into the appropriate request attribute for use by the <html:errors> tag, if any messages are required
void	saveErrors(HttpServletRequest request, ActionMessages errors)	Saves the specified error messages keys into the appropriate request attribute for use by the <html:errors> tag, if any messages are required

org.apache.struts.action.ActionForm

An ActionForm is a JavaBean optionally associated with one or more ActionMappings. Such a bean will have had its properties initialized from the corresponding request parameters before the corresponding Action.execute() function is called.

When the properties of this bean have been populated, but before the `execute()` function of the `Action` is called, this bean's `validate()` function will be called, which gives the bean a chance to verify that the properties submitted by the user are correct and valid. If this function finds problems, it returns an error messages object that encapsulates those problems, and Struts will return control to the corresponding input form. Otherwise, the `validate()` function returns `null`, indicating that everything is acceptable and the corresponding `Action.execute()` function should be called.

The `ActionForm` class *must* be subclassed in order to be instantiated.

Subclasses should provide property getter and setter methods for all of the bean properties they wish to expose. In addition, they should override any of the public or protected methods for which they wish to provide modified functionality.

Because `ActionForms` are JavaBeans, subclasses should also implement `Serializable`, as required by the JavaBean specification. Some containers require that an object meet *all* JavaBean requirements in order to use the introspection API upon which `ActionForms` rely.

Return Type	Function	Comments
`void`	`reset(ActionMapping mapping, HttpServletRequest request)`	Resets bean properties to their default state, as needed
`ActionErrors`	`validate(ActionMapping mapping, HttpServletRequest request)`	Validates the properties that have been set for this HTTP request, and returns an `ActionErrors` object that encapsulates any validation errors that have been found

org.apache.struts.upload.FormFile

This *interface* is used to represent a file uploaded by a client.

Return Type	Function	Comments
`void`	`destroy()`	Destroys all content for this form file
`String`	`getContentType()`	Gets the content type string for this file
`byte[]`	`getFileData()`	Gets the data in byte array for this file
`String`	`getFileName()`	Gets the filename of this file
`int`	`getFileSize()`	Gets the size of this file
`InputStream`	`getInputStream()`	Gets an `InputStream` that represents this file

org.apache.struts.tiles.ComponentContext

This class is the equivalent of `HttpServletRequest` for a Tile.

Return Type	Function	Comments
`void`	`addAll(Map newAttributes)`	Adds all attributes to this context
`void`	`addMissing(Map defaultAttrs)`	Adds all missing attributes to this context
`Object`	`getAttribute(String name)`	Gets an attribute from this context
`Iterator`	`getAttributeNames()`	Gets the names of all attributes
`void`	`putAttribute(String name,Object value)`	Puts a new attribute into the context

org.apache.struts.action.ExceptionHandler

An `ExceptionHandler` is a special class to handle exceptions generated in your `ActionForm` or `Action` subclasses. Refer to Chapter 9 on how to declare and use `ExceptionHandlers`.

Return Type	Function	Comments
`ActionForward`	`execute(Exception ex, ExceptionConfig ae, ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)`	Handles the `Exception`. Your subclasses must override this function.

APPENDIX C

Struts Tag Reference

This appendix is a reference on all Struts 1.2 tags, from the five tag libraries: HTML, Bean, Logic, Nested, and Tiles. We've included usage examples and information on Struts-EL and JSTL (see Chapter 10 for both), as well as JSF and Struts-Faces Integration Library (see Chapter 20 for both) equivalent tags where appropriate. This reference does *not* include the obsolete Template tag library, which has been superseded by the Tiles tag library.

To make this reference more compact, we've collected attributes that occur in more than one tag under one or more *common attribute sets*. These are introduced at the start of each section describing a tag library.

The HTML Tag Library

The custom tags in the HTML library are essentially in one-to-one relationship to the ordinary HTML `<form>` tag and its associated input tags, like the various `<input>` tags.

The purpose of this tag library is to enable you to connect the View tier to the Controller tier of your Struts webapp. The linkage between the two is done statically in `struts-config.xml`.

The tags in the HTML library can be divided into four groups based on function:

- **Form tags**: These tags transfer data from View to Controller. The tags are mostly in one-to-one correspondence with their HTML equivalents (`<form>` and the various HTML data input tags), but there are extra Struts-specific tags (e.g., `multibox`) that do much more to facilitate the easy transfer of data between View and Controller.

- **Message tags**: These tags display (in the View tier) messages that originate from the Controller.

- **URL tags**: These tags are counterparts to ordinary HTML tags that require a URL to function. The Struts versions exist primarily to allow you to use the names of global forwards instead of actual URLs.

- **Miscellaneous tags**: These tags have no clear functional grouping and have to be treated individually.

Tables C-1 to C-4 give a synopsis of these tags by their functional grouping. Tables C-1 to C-4 are based on the Apache documentation for those tags. A copy of the Apache License is available at `www.apache.org/licenses/LICENSE-2.0`.

Table C-1. *Form Tags of the HTML Tag Library*

Tag	Usage
`form`	Defines a form.
`checkbox`	Generates a checkbox input field.
`file`	Generates a file selection input field.
`hidden`	Generates a hidden field.
`multibox`	Generates multiple checkbox input fields.
`radio`	Generates a radio button input field.
`select` and `option`, `options`, or `optionsCollection`	`select` generates a drop-down list. The `option` elements are nested within `select`, and generate the options for the enclosing `select` element.
`text/password`	Generates a text/password input field.
`textarea`	Generates an HTML `textarea` input field.
`image`	Generates an image `input` field.
`button`	Generates a button input field.
`cancel`	Generates a cancel button.
`submit`	Generates a submit button.
`reset`	Generates a reset button.

Table C-2. *Message Tags of the HTML Tag Library*

Tag	Usage
`errors`	Displays error messages
`messages`	Iterates through error messages and messages (refer to the entry for `<logic:iterate>` for a description of these terms)

Table C-3. *URL Tags of the HTML Tag Library*

Tag	Usage
`base`	Generates an HTML `<base>` tag. This creates a reference from which all relative paths in your JSP will be calculated.
`img`	Generates an HTML `<img>` tag.
`link`	Generates an HTML hyperlink.
`rewrite`	Expands a given URI. Useful in creating URLs for input into your JavaScript functions.

Table C-4. *Miscellaneous Tags of the HTML Tag Library*

Tag	Usage
`html`	Generates an `<html>` tag. Also includes language attributes from the user's session.
`xhtml`	Tells other tags on the page to render themselves as XHTML 1.0–conformant tags.
`frame`	Generates an HTML frame.
`javascript`	Indicates the placement of autogenerated JavaScript. Used in conjunction with the Validator framework, described in Chapter 15.

Common Attribute Sets

There are attributes that are common to more than one tag of the HTML library. These may be naturally grouped into few *common attribute sets*, which we'll describe next. We'll give each set an abbreviation, which we'll use in subsequent descriptions of each tag.

The Event Handler Attribute Set

The biggest such attribute set consists of attributes corresponding to HTML event handler attributes. The Struts versions are exactly the same as the HTML versions. Let's refer to these as the *Event Handler attribute set,* or **evt-attrs** for short. The attributes in this set are described in Table C-5. The value of each attribute in this set is a JavaScript function that is called when a particular action occurs (e.g., mouse clicks, loss of focus, etc.). If you're familiar with HTML, Table C-5 should hold no surprises for you.

Table C-5. *The evt-attrs Attribute Set*

Attribute Name	Trigger
onblur	Element loses focus.
onchange	Element's value changed and element has lost focus.
onclick	Element is clicked.
ondblclick	Element is double-clicked.
onfocus	Element gains focus.
onkeydown	Key pressed when element has focus.
onkeypress	Key pressed then released when element has focus.
onkeyup	Key released when element has focus.
onmousedown	Mouse button is pressed while pointer is over element.
onmousemove	Mouse pointer is moved over element.
onmouseout	Mouse pointer leaves element.
onmouseover	Mouse pointer enters element.
onmouseup	Mouse button is released while pointer is over element.

The Accessibility Attribute Set

The next set is also HTML related, and these attributes aid in accessibility.

Table C-6 lists these tags, which we'll refer to as the Accessibility attribute set, or **acc-attrs** for short.

Table C-6. *The acc-attrs Attribute Set*

Attribute Name	Usage
accesskey	Specifies the key that if pressed, causes this element to gain focus.
alt	Specifies the alternate text for this element.
altKey	Specifies the message resource key for the alternate text.
tabindex	A positive integer that specifies the tab order for this element.
title	Specifies the advisory title for this element. In most graphical browsers, the advisory title appears as a tooltip when the mouse pointer goes over the element.
titleKey	Specifies the message resource key for the advisory title.

Most attributes in Table C-6 are simply mirror images of their HTML counterparts. The exceptions are `altKey` and `titleKey`, which use Struts' message resource file to produce a localized version of the actual attribute (`alt` and `title`, respectively).

The Rendering Attribute Set

The Rendering attribute set (**ren-attrs**) consists of attributes that specify the final rendering of the underlying HTML elements that correspond to a tag. These are listed in Table C-7.

Table C-7. *The ren-attrs Attribute Set*

Attribute Name	Usage
`disabled`	This boolean attribute disables (`disabled = "true"`) an input element. In most graphical browsers, this causes the HTML element to be grayed out.
`style`	Specifies the CSS styles for this element.
`styleClass`	Specifies the CSS stylesheet class for this element. `styleClass` corresponds to the `class` HTML attribute.
`styleId`	Specifies the HTML ID to be assigned to this element. That is, `styleId` corresponds to the `id` HTML attribute.

The Struts Attribute Set

The Struts attribute set (**struts-attrs**) consists of commonly used attributes specific to Struts. These are listed in Table C-8.

Table C-8. *The struts-attrs Attribute Set*

Attribute Name	Usage
`bundle`	Specifies the message resources bundle to use. If this isn't specified, the default `Application.properties` file is used. Bundles are a way for you to use multiple message resource files in your webapp. Note that the current support for this feature is uneven. Also, using it complicates maintenance. The `bundle` attribute is explained in more detail in the entry for `<bean:message>`. This attribute was newly introduced with Struts 1.2.5.
`indexed`	This boolean attribute is valid only when the enclosing tag is nested inside the `<logic:iterate>` tag. If `true`, the name of the rendered HTML tag will be indexed, for example, `myProperty[12]`. The number in brackets will be incremented for every iteration of the enclosing `<logic:iterate>` tag.
`property`	The name of the request parameter or HTML POST element that will be used when a form is submitted, set to the specified value (see the next entry).
`value`	Value of the input element. This value is submitted to the Controller tier, either in the request parameter or in the HTML POST itself.

The Initial Bean Attribute Set

The Initial Bean attribute set (**init-attr**) consists of just one attribute, `name`, which refers to a JavaBean whose corresponding property (specified by the `property` attribute of the element) is used as an initial value for the rendered element. The form bean associated with the enclosing form is the default bean for this task, and it is used if `name` is not specified.

The Error Style Attribute Set (Struts 1.2.5+)

The Error Style attribute set (**err-attrs**) was newly introduced with Struts 1.2.5. These attributes enable you to specify an error style if a given input element hits a validation error. Table C-9 describes these attributes.

Table C-9. *The err-attrs Attribute Set*

Attribute Name	Usage
`errorKey`	The key under which the error messages are stored. You only need to specify this attribute if you also specified the associated `<html:errors>`'s `name` attribute. The two must be the same.
`errorStyle`	The CSS style for the element if there is an error message for it.
`errorStyleClass`	The name of the CSS style class for the element if there is an error message for it.
`errorStyleId`	Assigns this new ID to the element, if there is an error message for it.

Struts-EL Tags for the HTML Tag Library

All tags in the HTML library have EL-enabled versions.

■ **Note** EL-enabled tags are those that allow you to use EL expressions. Refer to Chapter 10 for examples.

base

This tag is used to allow you to conveniently create an HTML `<base>` tag.

The HTML `<base>` tag allows you to specify the base URL from which to calculate relative paths (e.g., `../../myscripts.js` or `mypage.html`) in your JSPs. You might think that such URLs are resolved using the absolute path of the enclosing JSP page, but this isn't necessarily the case. The only way to guarantee this is to use the `<base>` tag, or much more conveniently, `<html:base>`.

Usage Restrictions

Both `<base>` and `<html:base>` must be a child tag of HTML's `<head>` tag.

Attributes

- `server`: Specifies the base URL. If you omit the `server` attribute, then Struts pastes the path to the enclosing JSP page. The `server` attribute gets rendered as the `href` attribute of `<base>`.
- `target`: If your app uses frames, then you can specify the name of the frame to which this base URL applies, in the page that defines the frame.

Examples

The simplest, and most useful, example is using the `<html:base />` tags by itself:

```
<head>
   <html:base />
</head>
```

The implied base is that of the JSP page itself. This gets rendered as

```
<head>
   <base href="http://www.mycompany.com/mywebapp/">
</head>
```

Similarly, if you wanted to fix the base of a frame within the current webapp, use this:

```
<head>
   <html:base target="myframe" />
</head>
```

If you need to set the base to point to another server, then use the `server` attribute:

```
<head>
   <html:base server="http://www.myothercompany.com" />
</head>
```

Equivalents

The Struts-Faces taglib (see Chapter 20) contains an `<s:base>` tag, which accepts a target attribute. There is no equivalent of the `server` attribute, though.

A hack for the `<html:base/>` tag is

```
<base href="<%= request.getScheme() %>://<%= request.getServerName() %>:
   <%= request.getServerPort() %>/
   <%= request.getContextPath() %>/">
```

button

<html:button> represents an HTML button. By itself, an <html:button> does nothing—it does not submit the form, for example. You have to use one (or more) of the event-handling atttibutes (see evt-attrs) to activate some JavaScript in order to do anything interesting.

■ **Note** Despite what the current Apache Struts online documents say, the value of the button is not sent as a URL parameter.

If you want to display a button defined by a graphic, then use <html:image> instead.

Usage Restrictions

This tag must be inside an <html:form> tag. You must specify the property attribute.

Attributes

The first four attribute sets (evt-attrs, acc-attrs, ren-attrs, and struts-attrs) are accepted. The only required attribute is property. If value isn't specified and if there is no text rendered in the body of the <html:button>, then the button text defaults to "Click".

Examples

This snippet will render an HTML button with the text label "Click", which calls the JavaScript function changeMode() when clicked:

```
<html:button onclick="changeMode(src)" property="button1" />
```

In this snippet, the imaginary JavaScript function might determine which button was clicked by using the test src.name == "button1".

The next example shows three buttons, all of which have the text "Click Me". The last button uses an <html:bean> to localize its text:

```
<html:button property="button1" value="Click Me" />
<html:button property="button2">Click Me</html:button>
<html:button property="button3">
   <html:bean message="msg.click-me"/>
</html:button>
```

Equivalents

The JSF <h:commandButton> is much more powerful, and allows you to easily link a button to a server-side function, using the action attribute:

```
<h:commandButton  action="#{myManagedBean.myFunction}" ...
```

Of course, for you to use this with Struts, you need to embed it within the Strut-Faces <s:form> tag. Refer to Chapter 20 for details on <h:commandButton> and <s:form>.

cancel

This tag displays a Cancel button on a form. Clicking this button causes the enclosing form to be submitted but simple validation is bypassed. However, the execute() function of the form's Action is called in order for the "next" page to be displayed. This means, of course, that the Action subclass must have special logic to determine if the Cancel button was clicked (see "Examples" in this section for details).

When the form data is submitted, the text appearing on the Cancel button is sent as a parameter on the request URL.

Usage Restrictions

This tag must be inside an <html:form> tag. You must *not* specify the property attribute. Doing so would cause the Cancel button not to be recognized as such by Struts.

Attributes

The first four attribute sets (evt-attrs, acc-attrs, ren-attrs, and struts-attrs) are accepted. However, do *not* specify the property attribute. If value isn't specified and if there is no text rendered in the body of the <html:cancel>, then the button text defaults to "Cancel".

Examples

The following example shows three buttons, all of which have the text "Cancel Me". The last Cancel button uses an <html:bean> to localize its text:

```
<html:cancel value="Cancel Me" />
<html:cancel>Cancel Me</html:cancel>
<html:cancel>
   <html:bean message="msg.cancel-me"/>
</html:cancel>
```

In order for you to detect a Cancel button being clicked, you'd use this code in your Action's execute():

```
public ActionForward execute(... HttpServletRequest request, ...){
   if (isCancelled(request)){
      //A Cancel button was pressed. Ignore form data
      return mapping.findForward("cancelled");
   }
   //normal form processing code here.
   ...
}
```

Equivalents

The JSF `<h:commandButton id="cancel" type="SUBMIT">` allows you to emulate the functionality of the `<html:cancel>` button. Of course, to use this with Struts, you need to embed it within the Strut-Faces `<s:form>` tag. Refer to Chapter 20 for details on `<h:commandButton>` and `<s:form>`.

checkbox

This tag renders a single check box input field within a form. See also `<html:multibox>`.

Usage Restrictions

This tag must be inside an `<html:form>` tag. The `property` attribute is required. The corresponding property on the `ActionForm` must be a boolean.

Attributes

All common attribute sets (evt-attrs, acc-attrs, ren-attrs, struts-attrs , init-attr, and the new err-attrs) are accepted, and have their usual meanings.

There are four important things to note:

- The corresponding property on the `ActionForm` subclass must be a boolean.
- The `value` attribute specifies the argument to put in `setXXX()` if the check box is checked. So, if you specify `value="true"` (or `on` or `yes`), then `setXXX(true)` is called if the check box is checked. If you use any other setting for value, then `setXXX(false)` is called if the check box is checked.
- The check box is rendered as checked only if `value` equals the return value of `getXXX()`. So, if `value="true"` (or `on` or `yes`) and `getXXX()` also returns `true`, then the check box is rendered as checked. Similarly, if value is not `true` (or `on` or `yes`) and `getXXX()` returns `false`, then the check box is also rendered as checked.

- The corresponding `setXXX()` function on the `ActionForm` subclass is only called if the check box is checked. This means that in order for your app to detect an unchecked check box, you need to call `setXXX()` in your `ActionForm`'s `reset()` function. `reset()` is automatically called by Struts before a form is (re)displayed, and before any `setXXX()` functions are called.

Examples

The code

```
<html:checkbox property="pregnant" />
<html:checkbox property="nonsmoker"  value="false" />
```

renders two check box fields. The corresponding `ActionForm` might be

```
public SurveyForm extends ActionForm{
   protected boolean _isPregnant, _isSmoker;

   public boolean getPregnant(){ return _isPregnant; }
   public void setPregnant(boolean isPregnant){
      _isPregnant = isPregnant;
   }

   public boolean getNonsmoker(){ return _isSmoker; }
   public void setNonsmoker(boolean isSmoker){
      _isSmoker = isSmoker;
   }

   public boolean isSmoker(){return _isSmoker;}

   public void reset(ActionMapping mapping,
                     HttpServletRequest request){
      //clear the checkboxes
      setPregnant(false);
      setNonsmoker(true);
    }
        ... //rest of SurveyForm
```

The `pregnant` property is straightforward: If the user checks the associated check box, then that translates directly to the `pregnant` property.

The `nonsmoker` property is less obvious: The underlying property that is stored is actually the opposite of `nonsmoker`. This just is a convenience, but as you can see, it comes at the price of added confusion.

An alternate, less confusing way of coding both the JSP form and `ActionForm` subclass would be

```
<html:checkbox property="pregnant" />
<html:checkbox property="nonsmoker"/>
```

Note the removed `value="false"`. The `ActionForm` subclass is now

```
public SurveyForm2 extends ActionForm{
   protected boolean _isPregnant, _isNonSmoker;

   public boolean getPregnant(){ return _isPregnant; }
   public void setPregnant(boolean isPregnant){
      _isPregnant = isPregnant;
   }

   public boolean getNonsmoker(){ return _isNonSmoker; }
   public void setNonsmoker(boolean isNonSmoker){
      _isNonSmoker = isNonSmoker;
   }

   public boolean isSmoker(){return !_isNonSmoker;}

   public void reset(ActionMapping mapping,
                     HttpServletRequest request){
      //clear the checkboxes
      setPregnant(false);
      setNonsmoker(false);
    }
        ... //rest of SurveyForm2
```

Equivalents

JSF's `<h:selectBooleanCheckbox>` is an equivalent. Enclose it in the Struts-Faces `<s:form>` element so that Struts processes the submitted value.

errors

This tag displays one or more error messages. If no matching error message is found, nothing is displayed. See also `<html:messages>`, which loops through error and ordinary messages.

Usage Restrictions

None.

Attributes

- `property`: The key with which to retrieve the error messages. Note that there might be more than one error message saved under the same property. If `property` isn't specified, then all error messages are displayed. This trick is very useful for debugging.

- `locale/bundle`: These attributes are used to specify a different locale object or message resource file. The `locale` specifies a key that can be used to look up the `Locale` object stored on the current session. The `bundle` attribute is explained in more detail in the entry for `<bean:message>`. You'd use these attributes to specify a locale different from the current user's locale or a message resource file different from the default one.

- `name`: The name of the object that holds error messages. Do not specify this attribute if you wish to use Struts' default error-reporting mechanism.

- `prefix/suffix`: Both are keys to text that will be rendered before and after each individual error message. For example, you might set

  ```
  myapp.errors.prefix=<font color=red>
  myapp.errors.suffix=</font>
  ```

  and you would use these attributes like this:

  ```
  <html:errors property="email"
               prefix="myapp.errors.prefix"
               suffix="myapp.errors.suffix" />
  ```

  and the error message would be displayed as

  ```
  <font color=red>Wrong email format</font>
  ```

- `header/footer`: These attributes are similar to `prefix/suffix`, but they apply to the start and end of a list of error messages. This happens when more than one error message is to be displayed.

Examples

Usually, you'd place an `<html:errors>` near an input element like so:

```
<html:text  name="contact" property="postcode" />
<html:errors property="postcode"/>
```

Equivalents

The `<s:errors>` tag from the Struts-Faces integration library is an equivalent.

file

This tag allows easy file uploading.

■ **Note** Chapter 11 contains a much fuller exposition on this useful tag.

Usage Restrictions

The `property` attribute is required. It must be a child tag of `<html:form>`.

Attributes

All common attribute sets (evt-attrs, acc-attrs, ren-attrs, struts-attrs , init-attr, and the new err-attrs) are accepted, and have their usual meanings.

Besides these, there are a few attributes specific to `file`:

- `accept`: A comma-delimited list of "acceptable" file extensions. These are displayed by the browser. Unfortunately, most browsers ignore this information.
- `size`: The length of the displayed field. If omitted, the decision is left to the browser.
- `maxlength`: The maximum number of characters to accept. The default is to allow an unlimited number of characters. Most web browsers ignore this attribute.

Examples

Please refer to Chapter 11 for examples.

Equivalents

There is no pure-JSF equivalent, but MyFaces sports an extension called `<x:inputFileUpload>` that does the trick. The mechanism used to access the uploaded file is very similar to Struts' `FormFile` (it's called `UploadedFile` instead). However, this leaves you in a bit of a bind because the current Struts-Faces integration library (1.0) doesn't work with MyFaces.

form

Represents an HTML form. It is also the mandatory parent element for many input tags.

Usage Restrictions

The `action` attribute is required.

Attributes

Besides the ren-attrs attributes, `form` accepts the following attributes:

- `action`: The name of the form handler that handles processing for this form's data. This attribute is required.
- `method`: The HTTP submission for this form—either POST or GET, with POST being the default.
- `focus`/`focusIndex`: `focus` specifies the name of the field on the form that should take focus when the form first loads. Struts autogenerates the JavaScript for this to happen. If the `focus` attribute isn't specified, then no such JavaScript is autogenerated. `focusIndex` applies to input elements that are indexed. You can specify the index of the element that takes focus.
- `enctype`: Specifies the encoding used when submitting this form's data. You only need to set this when you have an `<html:file>` input element in the form (see the corresponding entry for more details, or refer to Chapter 11).
- `onsubmit`/`onreset`: These are the names of JavaScript event handlers to invoke when the form is submitted or reset.
- `acceptCharset`: Character encodings that are valid for this form (since Struts 1.2.7). This is a standard HTML attribute.
- `readonly`: If `true`, means the form data may not be edited.
- `scriptLanguage`: If set to `false`, omits the `language` attribute in the generated `<script>` tag. This attribute is ignored if you've specified XHTML rendering.
- `target`: The name of the window to which this form's data is submitted. This is a standard HTML attribute.

Examples

Most commonly, you'd specify the `action` and `focus` attributes:

```
<html:form  action="MyFormHandler.do" focus="email">
 Key in your email address: <html:text  property="email"/>
 ...
</html:form>
```

Equivalents

The Struts-Faces `<s:form>` is the best replacement for `<html:form>`. Refer to Chapter 20 for details.

frame

Renders an HTML `<frame>`. The advantage of using `<html:frame>` is that you can use global forwards or form handlers to easily put content into your frames.

Usage Restrictions

You must specify either `action/module`, `href`, `page`, or `forward`.

Attributes

- `action/module`, `href`, `page`, or `forward`: Used to specify a URL for the frame. The `action/module` pair specifies a form handler. The `action`, of course, must begin with a slash, and if you want to specify a module, use the `module` attribute. `href` is either a relative or absolute URL. `page` is a module-relative URL (and therefore, must begin with a slash). `forward` is the name of a global forward.

- `anchor`: An optional HTML anchor for the link.

- `paramName`,`paramProperty`,`paramScope` and `paramId`: You use these to create a single request parameter. The request parameter is appended to the final URL. The first set of three attributes is used to locate a single object on the current request or session. This object's `toString()` is the single parameter value. The name of the parameter is given by `paramId`. Refer to `<html:link>` for an example.

- `name/property/scope`: You use these to create multiple request parameters. The request parameters are appended to the final URL. These attributes are used to locate an object of type `java.util.Map`. The `Map`'s keys are the parameter names, and the corresponding values are the parameter values. If you specify `property`, you must also specify `name`. Refer to `<html:link>` for an example.

- `transaction`: If true, appends the current transaction token to the URL as a request parameter. Refer to the entry for `<logic:present>` for details on transaction tokens.

- `title/titleKey`: Refer to acc-attrs for details on these.

- `style/styleClass/styleId`: Refer to ren-attrs for details on these.

- `scrolling`: Specifies if scrollbars are to be created when necessary (`scrolling="auto"`), or always (`yes`) or never (`no`).

- marginheight/marginwidth: Specifies the height of the top and bottom margins (marginheight) or the left and right margins (marginwidth). The units used are pixels.
- frameborder: Specifies if a border needs to be rendered (frameborder="1") or not (frameborder="0").
- frameName: The name of the rendered <frame>.
- noresize: A boolean attribute indicating if this frame can be resized (false) or if its size is fixed (true).
- longdesc: The URL to a longer description for this frame. This attribute is a standard HTML one, and you should consult an HTML reference for details.
- bundle: Allows you to select a message resource file different from the default one. The bundle attribute is explained in more detail in the entry for <bean:message>.

Examples

Here's a page with two frames:

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<frameset rows="10%,*">
  <html:frame action="/NavBar.do" scrolling="no" frameName="navbar" />
  <html:frame page="/index.jsp" scrolling="yes" frameName="main" />
</frameset>
```

Equivalents

None.

hidden

This tag represents a hidden form field.

Usage Restrictions

The property attribute is required. It must be a child tag of <html:form>.

Attributes

The first five common attribute sets (evt-attrs, acc-attrs, ren-attrs, struts-attrs, and init-attr) are accepted, and have their usual meanings.

Besides these, there is the write attribute, which if set to true causes the value of the hidden field to be displayed. This is useful for debugging.

Examples

The code

```
<html:hidden  property="clientId" />
```

specifies a hidden field that holds the value for the property called `clientId` on the current form bean.

You can also specify a `value` attribute like so:

```
<html:hidden  property="command" value="update" />
```

Your `Action` subclass that reads the form data can then dispatch on the hidden value. Look at the entry for `<bean:define>` for another useful example.

Equivalents

JSF's `<h:inputHidden>` is an equivalent for `<html:hidden>`, but you need to enclose it within the Struts-Faces `<s:form>` tag so that the associated form bean is populated with the submitted value:

```
<s:form ...
   <!-- equivalent to html:hidden -->
   <h:inputHidden  id="clientId" binding="#{MyFormBean.clientId}" />
   ...
</s:form>
```

html

This tag inserts `<html>` and `</html>` tags, with `language` attributes from the current user's locale.

Usage Restrictions

None—if you use it, other Struts tags should be nested within it.

Attributes

- `xhtml`: If set to `true`, causes nested Struts tags to render themselves as XHTML 1.0. (See also `<html:xhtml>`.)

- `lang`: (since Struts 1.2.0) . If set to `true`, puts in a `lang` attribute in the final `<html>` tag. The value is searched for in the following order: (1) The current session's `Locale`, (2) the `accept-language` HTTP request header, and (3) the server's default locale. Consult an HTML reference to understand how to use HTML's `lang` attribute.

Examples

Here's how to render an XHTML 1.0–conformant page:

```
<html:html  xhtml="true">
   <!-- your Struts tags here -->
   ...
</html:html>
```

Equivalents

The `<s:html>` tag from the Struts-Faces integration library is an equivalent.

image

This tag renders an HTML image input field. Clicking on the displayed image causes the containing form to be submitted.

Usage Restrictions

This tag must be inside an `<html:form>` tag. You need to specify `src`, `srcKey`, `page`, or `pageKey`.

Attributes

The first four common attribute sets (evt-attrs, acc-attrs, ren-attrs, and struts-attrs) are accepted, and have their usual meanings, except for the `property` attribute (explained in a bit).

There are a few attributes specific to `image`:

- `property`: This specifies the function on the base object that returns a JavaBean, which has `setX(int x)` and `setY(int y)` functions, to save the (x,y) coordinate of the mouse click for the image input field. If you want to use `setX()` and `setY()` on the base object, simply use `property=""`.

- `src/srcKey/page/pageKey`: You use one of these to specify the image to be used. `src` is the URL for the image. `page` is the module-relative URL for the image (and therefore, must begin with a slash). The `key` versions are message resource keys that point to the actual URL.

- `border`: The width of the border around the displayed image. The units are pixels.

- `locale`: Specifies a key that can be used to look up the `Locale` object stored on the current session. This object is used to determine the value of any keys used. If omitted, the default Struts message resource file is used.

Examples

The code that follows shows an image input field in your JSP:

```
<html:form ...>
   <html:image page="/myMapImage.jpg" property="" />
</html:form>
```

The `ActionForm` subclass might be

```
public MapActionForm extends ActionForm{
   public final int UNKNOWN = Integer.MAX_VALUE;
   protected int _x , _y;

   public int getX(){ return _x; }
   public void setX(int x){ _x = x; }

   public int getY(){ return _y; }
   public void setY(int y){ _y = y; }

   public void reset(ActionMapping mapping,
                     HttpServletRequest request){
       _x = _y = UNKNOWN;
   }
}
```

Equivalents

None.

img

This tag renders an HTML `<img>`.

Usage Restrictions

You must specify one of these attributes: `action`, `module` or `src/srcKey`, or `page/pageKey`.

Attributes

The first three common attribute sets are accepted: evt-attrs, acc-attrs, and ren-attrs. Besides these, there are a few attributes specific to `img`:

- action, module or src/srcKey, or page/pageKey: Used to locate the image to display. The first action/module pair specifies a form handler. The action, of course, must begin with a slash, and if you want to specify a module use the module attribute. src is the URL for the image. page is the module-relative URL for the image (and therefore, must begin with a slash). The key versions are message resource keys that point to the actual URL. They are very useful to display localized images.

- border: Specifies the width of the border around the displayed image. The units are pixels.

- locale/bundle: These are used to specify a different Locale object or message resource file. locale specifies a key that can be used to look up the Locale object stored on the current session. The bundle attribute is explained in more detail in the entry for <bean:message>.

- align: Specifies the alignment of the image relative to surrounding text. This is a standard HTML attribute for the <img> element. Valid values are left, right, top, texttop, bottom, absbottom, middle, and absmiddle. Consult an HTML reference to understand what these settings do.

- height/width: Specifies the height and width of the finally displayed image in pixels. The resizing of the image is done by the browser.

- hspace/vspace: The amount of horizontal space (hspace) and vertical space (vspace) between the image and surrounding elements. The units are pixels.

- imageName: Renders a name attribute in the <img> tag: name=<imageName>. Used for scripting.

- ismap: This boolean attribute is the counterpart of the ismap attribute of HTML's <img> tag. It's used to indicate a server-side image map—which is used only for ancient web browsers that don't understand the HTML <map> element. For this reason, you are extremely unlikely to use ismap.

- usemap: The name of the HTML <map> element to use. This turns the image into a clickable image map. Refer to an HTML reference on how to create a <map>. The browser's behavior when the user clicks the image is entirely processed on the client. No server-side processing is involved. If you want to pass mouse click positions to server-side code, use <html:image> instead.

- `paramName`,`paramProperty`,`paramScope` and `paramId`: You use these to create a single request parameter. The request parameter is appended to the final URL for the image. The first set of three attributes is used to locate a single object on the current request or session. This object's `toString()` is the single parameter value. The name of the parameter is given by `paramId`.

- `name`/`property`/`scope`: You use these to create multiple request parameters. The request parameters are appended to the final URL for the image. These attributes are used to locate an object of type `java.util.Map`. The `Map`'s keys are the parameter names, and the corresponding values are the parameter values. If you specify `property`, you must also specify `name`.

- `useLocalEncoding`: If set to `true`, tells Struts to use whatever the character encoding is for the current `HttpServletResponse`.

Examples

Here's how to render a localized image:

```
<html:img  srcKey="app.images.companylogo" />
```

To understand how to create single or multiple request parameters on the image's URL, refer to the entry for `<html:link>`.

Equivalents

JSF's `<h:graphicImage>` is an equivalent.

javascript

This tag causes the generation of JavaScript for client-side validation. This tag is very closely tied to the Validator framework.

Usage Restrictions

Using this tag only makes sense when it's accompanying an `<html:form>`, since the generated JavaScript is used for client-side form validations. The JavaScript code is autogenerated by the Validator framework. Therefore, your `ActionForm` subclasses must be subclasses of `ValidatorForm` as well in order for this tag to work.

Attributes

The most common use of this tag is without any attributes defined. However, you can customize the generated JavaScript a little by using the following attributes:

- bundle: Specifies a message resource bundle (see Table C-8) to use in generating error messages. This attribute was added in Struts 1.2.7. See also the entry for <bean:message>.

- htmlComment: If set to true, enclose the JavaScript with HTML comment delimiters (<!-- and -->). This is needed to prevent very old browsers from rendering JavaScript on to the page. This setting is ignored if you've used <html:xhtml> or <html:html xhtml="true">. The default is true.

- cdata: If cdata="true", and if you've used <html:xhtml> or <html:html xhtml="true">, the generated JavaScript is enclosed by XML's CDATA[...] delimiter.

- method: Specifies a different prefix for the names of the autogenerated JavaScript. Use this attribute to prevent name clashes between your own JavaScript and the Struts-generated JavaScript code.

- scriptLanguage: If set to false, omits the language attribute in the generated <script> tag.

- staticJavascript/dynamicJavascript: Both of these take boolean values. staticJavascript (default is true) specifies whether the static JavaScript code from the Validator framework needs to be pasted into the page. Similarly, dynamicJavascript (the default is true) specifies if the dynamic JavaScript code generated for the form should be pasted into the page. Ordinarily, you'd leave out both these attributes.

- src: This puts an HTML src attribute into the rendered <script> tag. You'd use this to specify your own JavaScript files you want loaded when the page is displayed by the browser.

- page/formName: The Validator framework allows you to create multipage validations (this topic is not covered in this book). This feature is useful in creating multipage wizards that simplify filling in complex data. Typically, the user is presented with one subform at a time, each with "next" and "back" buttons. The page attribute is the number of the current page, and the formName attribute is the name of the form bean.

Examples

The most common usage is to require that all validations be made. You do this by specifying no attributes:

```
<html:javascript />
```

This would work if pasted anywhere on the page.

Equivalents

The <s:javascript> tag from the Struts-Faces integration library is an equivalent.

link

This tag renders an HTML link.

Usage Restrictions

You must specify either action/module, href, page, forward, or linkName.

Attributes

The first three common attribute sets are accepted: evt-attrs, acc-attrs, and ren-attrs. Besides these, there are a few attributes specific to link:

- action/module, href, page, forward, linkName: Used to specify a URL for the link. The action/module pair specifies a form handler. The action, of course, must begin with a slash, and if you want to specify a module, use the module attribute. href is the URL for the link. page is the module-relative URL for the link (and therefore must begin with a slash). forward is the name of a global forward. linkName is the name of another link on the same page. The URL for that link is used for this one.

- anchor: An optional HTML anchor for the link.

- bundle: Used to specify a different message resource file. The bundle attribute is explained in more detail in the entry for <bean:message>.

- transaction: If true, appends the current transaction token to the link's URL as a request parameter. Refer to the entry for <logic:present> for details on transaction tokens.

- paramName,paramProperty,paramScope and paramId: You use these to create a single request parameter. The request parameter is appended to the final URL for the link. The first set of three attributes is used to locate a single object on the current request or session. This object's toString() is the single parameter value. The name of the parameter is given by paramId.

- name/property/scope: You use these to create multiple request parameters. The request parameters are appended to the final URL for the link. These attributes are used to locate an object of type java.util.Map. The Map's keys are the parameter names, and the corresponding values are the parameter values. If you specify property, you must also specify name.

- useLocalEncoding: If set to true tells Struts to use whatever the character encoding is for the current HttpServletResponse.

Examples

Here's a simple example using a global forward:

```
<html:link forward="success" />
```

The rendered link will have the true location for the success global forward. You can also include request parameters for the link:

```
<html:link page="/mypage.jsp" paramName="myRequestValue"
        paramId="action" />
```

This snippet will render a link with a request parameter named action, whose value is given by the value of toString() called on the bean myRequestValue.

Here's how you'd create multiple request parameters:

```
<html:link href="http://www.kenyir.org/index.jsp" name="myMap"/>
```

Equivalents

The <s:commandLink> tag from the Struts-Faces integration library is an equivalent. You may also use JSF's <h:commandLink>, but you lose the ability to use global forwards and form handlers.

messages

messages is an iterator for *error messages* and *messages*—please refer to the entry for <logic:messagesPresent> for an explanation of these two terms. By default, <html:messages> is an iterator over error messages, but you can iterate over messages by setting the messages attribute to true.

By itself, <html:messages> does not print out either error messages or messages. You have to nest a suitable tag within it (e.g., <bean:write>) to do this.

Usage Restrictions

The id attribute is required.

Attributes

- id: This behaves just like the id attribute of <logic:iterate>—it exposes a variable that refers to a single message.

- message: A boolean attribute that indicates whether to iterate over error messages (false, or omitted) or messages (true).

- property: The key with which to retrieve messages. Note that there might be more than one message saved under the same property. If property isn't specified, then all messages are displayed. This trick is very useful for debugging.

- name: The name of the object that holds messages. Do not specify this attribute if you wish to use Action's saveErrors() or saveMessages().

- header/footer: Both are message resource keys to text that will be rendered at the start and end of the iteration. This happens only when there is more than one error message to be displayed. Refer to the entry for <html:errors> for details.

- locale/bundle: These are used to specify a different Locale object or message resource file. locale specifies a key that can be used to look up the Locale object stored on the current session. The bundle attribute is explained in more detail in the entry for <bean:message>.You'd use these attributes to specify a locale different from the current user's locale or a message resource file different from the default one.

Examples

Here's a simple example that displays all error messages:

```
<html:messages  id="anError">
   <bean:write  name="anError" />
</html:messages>
```

Remember, error messages are present either from simple validation failure (see Chapter 6) or registered using Action's saveErrors() (see Chapter 7).

Here's how to display messages:

```
<html:messages  id="aMessage"  message="true">
   <bean:write  name="aMessage" />
</html:messages>
```

Messages are created in exactly the same way as error messages, but are registered using Action's saveMessages(), which has the same signature as saveErrors().

Here's how to display a header and footer (if there are any error messages to display):

```
<html:messages  id="anError" header="err.header" footer="err.footer">
   <bean:write  name="anError" />
</html:messages>
```

This assumes that the message resource keys err.footer and err.header have been defined in your Application.properties file:

```
err.header=<b>
err.footer=</b>
```

So the HTML bold tag (`<b>`) is printed at the start and end of the iteration.

Equivalents

There are none, but it's easy to create an ugly hack: Error messages are stored under the key `Globals.ERROR_KEY` and messages are stored under `Globals.MESSAGE_KEY`. Both are stored on the request. `Globals` is a Struts class (`org.apache.struts.Globals`) that contains various constants. The stored object is either an `ActionMessages` or `ActionErrors` instance. You can easily create a reference to this object using JSTL's `<c:set>`, and use the defined messages object in the page.

multibox

This tag renders one or more check box input fields, based on an underlying array.

See also `<html:checkbox>`. Note, however, that unlike `<html:checkbox>`, the corresponding property on the `ActionForm` need not be a boolean. Any type of array will do.

Usage Restrictions

This tag must be inside an `<html:form>` tag. The `property` attribute is required.

Attributes

All common attribute sets (evt-attrs, acc-attrs, ren-attrs, struts-attrs, init-attr, and the new err-attrs) are accepted, and have their usual meanings.

A few important points to note:

- The return value can be specified either by the `value` attribute or the content nested in the body of the tag.
- To use this tag, you'd usually nest it within a `<logic:iterate>` (or any other compatible looping construct like JSTL's `<c:forEach>`), and use scriptlets or the EL-enabled version of this tag to dynamically set the `value` attribute (if you need to nest the content instead, then you must use scriptlets).
- The values are submitted only if a check box is checked. This means that in order to detect a cleared check box, you'd need to set the underlying array to a zero-length array. See the following examples for more on this.

Examples

Consider the <html:multibox> on a form:

```
<html-el:form ...>
   <logic:iterate  name="contacts"  id="contact">
     <html-el:multibox  property="ids" value="${contact.getId()}" />
   </logic:iterate>
</html-el:form>
```

and the associated ActionForm subclass:

```
public SelectedContactIds extends ActionForm{
   protected String[] _ids;

   public void setIds(String[] ids){ _ids = ids; }

   public String[] getIds(){return _ids;}

   public void reset(ActionMapping mapping,
                     HttpServletRequest request){
      // We use a new String array instead
      // of just null to allow easy migration to
      // the Validator framework, which requires
      // a non-null value for arrays.

      _ids = new String[0];
   }
}
```

The length of the _ids variable isn't fixed—it depends on how many of the rendered check boxes were checked.

Equivalents

None.

radio

This tag renders a radio button input field.

Usage Restrictions

This tag must be inside an <html:form> tag. The property and value attributes are required.

Attributes

All common attribute sets (evt-attrs, acc-attrs, ren-attrs, struts-attrs , init-attr, and the new err-attrs) are accepted, and have their usual meanings.

In addition to these, there is the `idName` attribute, which points to a JavaBean. When `idName` is specified, the `value` attribute is used as a property name for this JavaBean. The combination is used to specify a return value for the radio button.

Examples

Here's an example of a few radio buttons on a form. Notice how all radio buttons refer to the same property (`color`), thereby allowing the user to select one color from a list:

```
<html:radio property = "color"   value="Red"/>Red<br>
<html:radio property = "color"   value="Yellow"/>Yellow<br>
<html:radio property = "color"   value="Blue"/>Blue<br>
```

Here's the same example but using the `idName` attribute:

```
<logic:iterate  name="colors"  id="c">
   <html:radio idName="c" value="color"/>
   <bean:write   name="c" property="color"/><br>
</logic:iterate>
```

In this second example, `colors` is an iteratable object (see the entry for `<logic:iterate>`) and holds JavaBeans (exposed in the snippet by the variable `c`), which have a `getColor()` function.

Equivalents

JSF's `<h:selectOneRadio>` is an equivalent.

reset

This tag displays a button that if clicked, causes the enclosing form's fields to be cleared.

Usage Restrictions

This tag should be inside an `<html:form>` tag.

Attributes

The first four attribute sets (evt-attrs, acc-attrs, ren-attrs, and struts-attrs) are accepted. If `value` isn't specified and if there is no text rendered in the body of the `<html:reset>` tag, then the button text defaults to "Reset".

Examples

This example shows three submit buttons, all of which have the text "Reset Me". The last reset button uses an <html:bean> tag to localize its text:

```
<html:reset value="Reset Me" />
<html:reset >Reset Me</html:reset >
<html:reset >
   <html:bean message="msg.reset-me"/>
</html:reset >
```

In all cases, if the user clicks the button, the form gets submitted, the associated ActionForm's reset() function is invoked, and the page is then redisplayed. Your ActionForm subclasses might override this function to provide custom resetting of the form's fields.

Equivalents

The JSF's <h:commandButton id="reset" type="RESET"> allows you to emulate the functionality of the <html:reset> button. Of course, for you to use this with Struts, you need to embed it within the Strut-Faces <s:form> tag. Refer to Chapter 20 for details on <h:commandButton> and <s:form>.

rewrite

This tag resolves and renders a URL. The rules used are similar to those for <html:link>. Unlike <html:link>, though, the URL isn't embedded within an HTML <a> tag; the URL is rendered by itself. You might find this useful for debugging or for use in scripts.

Usage Restrictions

You must specify either action/module, href, page, or forward.

Attributes

- action/module, href, page, or forward: Used to specify a URL. The action, module pair specifies a form handler. The action, of course, must begin with a slash, and if you want to specify a module, use the module attribute. href is either a relative or absolute URL. page is a module-relative URL (and therefore, must begin with a slash). forward is the name of a global forward.

- anchor: An optional HTML anchor for the link.

- transaction: If true, appends the current transaction token to the URL as a request parameter. Refer to the entry for <logic:present> for details on transaction tokens.

- paramName,paramProperty,paramScope and paramId: You use these to create a single request parameter. The request parameter is appended to the final URL. The first set of three attributes is used to locate a single object on the current request or session. This object's toString() is the single parameter value. The name of the parameter is given by paramId.

- name/property/scope: You use these to create multiple request parameters. The request parameters are appended to the final URL. These attributes are used to locate an object of type java.util.Map. The Map's keys are the parameter names, and the corresponding values are the parameter values. If you specify property, you must also specify name.

- useLocalEncoding: If set to true, tells Struts to use whatever the character encoding is for the current HttpServletResponse.

Examples

Here's a simple example using a global forward:

```
<html:rewrite forward="success" />
```

The rendered string will be the true URL for the success global forward. For more examples, refer to the entry for <html:link>.

Equivalents

None.

select, with option, options, and optionsCollection

select displays an HTML selection.

The other option tags are nested within it in order to render options for the enclosing select element:

- option: Renders a single option. There are attributes you can use to help with localizing the displayed text.

- options: Displays a list of options, populated from a JavaBean.

- optionsCollection: A more streamlined version of options. Also displays a list of options populated from a JavaBean.

Each rendered <option> has a value and a label. The value is the value submitted if that option is selected. The label is the text presented to users in order for them to make the selection.

Usage Restrictions

The `select` tag must be inside an `<html:form>` tag, and the various `option` tags must be nested within an `<html:select>` tag. There are a couple other restrictions:

- The `property` attribute is required for `select`.
- The `value` attribute is required for `option`.

Attributes for select

All common attribute sets (evt-attrs, acc-attrs, ren-attrs, struts-attrs , init-attr, and the new err-attrs) are accepted, and have their usual meanings. Besides these, there are a couple of attributes specific to `select`:

- `multiple`: Enables multiple selections if specified (the actual value is unimportant). The underlying property must be an array if the `multiple` attribute is specified. As with `<html:multibox>`, you must set the underlying property to a zero-length array in your `ActionForm`'s `reset()` if you want to detect an unselected item. See the entry for `<html:multibox>` for details.
- `size`: The number of options to display at once.

Attributes for option

The `option` tag accepts the ren-attrs attribute set. Besides this, the following attributes are specific to `option`:

- `value`: The submitted value. This attribute is required.
- `key`: The message resource key that specifies the label for the option. If omitted, the label is the text nested within the `option` tag's body.
- `locale/bundle`: These attributes are used to specify a different `Locale` object or message resource file. `locale` specifies a key that can be used to look up the `Locale` object stored on the current session. The `bundle` attribute is explained in more detail in the entry for `<bean:message>`. You'd use these attributes to specify a locale different from the current user's locale or a message resource file different from the default one.

Attributes for options

The `option` tag accepts the ren-attrs attribute set, except for the `styleId` attribute. (This attribute is left out because the `options` tag might render more than one HTML `<option>` element, and the HTML standard forbids you from having two elements with the same `id`.)

There are two ways to specify the values and labels to use for each rendered <option> element:

- Specify the collection/property/labelProperty attributes: The collection attribute gives the name of a collection of JavaBeans, each holding the value and label for a single <option>. The property attribute is the name of the getXXX() function that is called on each bean to obtain the value for an <option>. Similarly, the labelProperty attribute is the name of the getXXX() function that is called on each bean to obtain the label for an <option>. If labelProperty is omitted, the <option>'s value is used as a label.

- Specify name/property and labelName/labelProperty attributes: The name and property pair specifies a collection in any scope from which to get the values for each rendered <option>. If you specify the property attribute only, then the ActionForm subclass is the implicit base object. Similarly, the labelName and labelProperty pair specifies a collection from which to get the labels for each rendered <option>. If only labelProperty is specified, then the ActionForm subclass is the implicit base object. If neither labelName nor labelProperty is specified, the labels are the same as the values.

The remaining attribute of <html:options> is filter, which if true filters the labels for HTML reserved characters (like <, >, or &). The default is true.

Attributes for optionsCollection

optionsCollection is a simplified, streamlined version of <html:options>. Like <html:options>, the optionsCollection tag accepts the ren-attrs attribute set, except for the styleId attribute. It also has the filter attribute, like <html:options>.

optionsCollection assumes that the values and labels for each rendered <option> are stored in a collection of JavaBeans. This collection is specified using the name and property attributes. If you specify the property attribute only, then the ActionForm subclass associated with the enclosing form is the implicit base object. Each JavaBean is assumed to have two getXXX() functions from which to obtain the value and label for a single <option> tag.

By default, the name of the function for obtaining values is getValue(). The function for labels is getLabel(). If you wish to specify different names for each, then use the value and label attributes. For example, setting value="color" would mean that the getColor() function is called to determine the value for each <option>.

The LabelValueBean Class

For options and optionsCollection, you need to have a JavaBean class with getXXX() functions for values and labels. Struts has a utility class called org.apache.struts.utility.LabelValueBean that does exactly this. It has the functions getValue() and

getLabel() that are ready to use with optionsCollection. Listing C-1 shows how you might use this class (perhaps in an Action subclass).

Listing C-1. *Using the LabelValueBean Class*

```
//create collection to hold value-label pairs
List labelValuePairs = new ArrayList();

//populate collection
labelValuePairs.add( new  LabelValueBean("labelOne", "valueOne"));
labelValuePairs.add( new  LabelValueBean("labelTwo", "valueTwo"));

//save to request under suitable label
request.setAttribute("myOptions", labelValuePairs);
```

Simple Performance Optimizations

For performance reasons, you should save and reuse a single collection instance, and not re-create it in each populating Action.

Sometimes, though, your selection may require a mix of dynamic and static options. One simple solution in this situation is to use more than one <html:optionsCollection>—one for the static options that come from a cache, and another that refers to dynamically created value-label pairs.

Even in the case of dynamically created pairs, some optimization is possible. The most time-consuming task is creating the new LabelValueBean objects. If the possible value-label combinations are finite, you might want to cache them in a WeakHashMap, which acts as a temporary cache. Of course, the benefits of caching the JavaBeans this way might be outweighed by the performance hit in calculating the keys to retrieve the bean from the WeakHashMap. So, this technique works best when you already have a usable key at hand.

Internationalization

The use of JavaBeans instead of Struts' message resource mechanism means that the option labels can't be readily localized. One simple solution is to use a custom class (see Listing C-2) for the collection (we'll use an Iterator), and to use LabelValueBeans to store message resource keys pointing to the localized label instead of the actual label.

The translation from message key to actual label while at the same time not creating unnecessary objects requires some trickery, as you can see from Listing C-2.

Listing C-2. *The LocalizableIterator Class*

```
/************************************************
* Copyright 2005 Arnold Doray
* This code is released under the Lesser GNU
* General Public License. Please refer to:
* http://www.gnu.org/copyleft/lesser.html
* for details on this license.
************************************************/
import java.util.Iterator;
import java.util.Locale;
import org.apache.struts.util.MessageResources;
import org.apache.struts.util.LabelValueBean;

public class  LocalizableIterator implements Iterator{

   protected LocalizedLabelValueBean _bean = null;
   protected Iterator _labelValuePairs = null;

   public  LocalizableIterator(){
      _bean = new LocalizedLabelValueBean();
   }

   public void set(Iterator labelValuePairs, Locale locale,
                   MessageResources resources){
      _labelValuePairs =  labelValuePairs;
      _bean.setLocale(locale);
      _bean.setResources(resources);
   }

   public boolean hasNext(){
      return _labelValuePairs.hasNext();
   }

   public Object next(){
      return _bean.setLabelValueBean(
                     (LabelValueBean)_labelValuePairs.next());
   }
```

```
    public void remove() throws  UnsupportedOperationException{
       throw new UnsupportedOperationException();
    }

    public class LocalizedLabelValueBean{
      private MessageResources _resources;
      private Locale _locale;
      private LabelValueBean _delegate;
      private String _label = null;

      private void setLocale(Locale locale){
         _locale = locale;
      }
      private void setResources(MessageResources resources){
         _resources = resources;
      }
      private LocalizedLabelValueBean
                 setLabelValueBean(LabelValueBean delegate){
         _delegate =  delegate;
         _label = null;
         return this;
      }

      public String getValue(){
         return _delegate.getValue();
      }

      public String getLabel(){
         if(null == _label){
            _label = _resources.getMessage(_locale,
                                           _delegate.getLabel());

         }
         return _label;
      }
    }
}
```

As Listing C-2 shows, the `LocalizableIterator` produces the same object when `next()` is called, each time configured with a different `LabelValueBean`. Here's how you might use this class in your `Action` subclasses. Remember that, unlike Listing C-1, the underlying

collection of LabelValueBean (given by the variable labelValuePairs in Listing C-2) now stores message resource keys instead of actual labels:

```
//somehow get collection of LabelValueBeans
//the labels on these are really message resource keys
List labelValuePairs = ...

//create a new LocalizableIterator
LocalizableIterator li = new LocalizableIterator();

//initialize it
li.set(labelValuePairs.iterator(),
       getLocale(request), getResources(request));

//save to request under suitable label
request.setAttribute("myOptions", li);
```

Examples

You may mix one or more option/options/optionsCollection tags in your JSP code. Here's an example:

```
<html:select  property="selectedColor">
   <html:option  value="new-color" key="app.prompt.newcolor" />
   <html:optionsCollection name="myOptions" />
</html:select>
```

Or, to enable multiple selections, use this:

```
<html:select  property="selectedColor" multiple="true">
   <html:optionsCollection name="myOptions" />
   <html:options  collection = "myOtherOptions"
                  property = "value" labelProperty="label" />
</html:select>
```

Remember, the value of the multiple attribute is unimportant—as long as the multiple attribute is specified, Struts assumes the underlying property is an array.

Equivalents

JSF's <h:selectOneListbox> is an equivalent for a single-selection list. <h:selectManyListbox> is an equivalent for a multiple-selection list.

submit

This tag displays a button that if clicked causes the enclosing form to be submitted for processing.

Usage Restrictions

This tag must be inside an `<html:form>` tag.

Attributes

The first four attribute sets (evt-attrs, acc-attrs, ren-attrs, and struts-attrs) are accepted. If `value` isn't specified and if there is no text rendered in the body of the `<html:submit>`, then the button text defaults to "Submit".

Examples

This example shows three submit buttons, all of which have the text "Submit Me". The last submit button uses an `<html:bean>` to localize its text:

```
<html:submit value="Submit Me" />
<html:submit >Submit Me</html:submit>
<html:submit >
   <html:bean message="msg.submit-me"/>
</html:submit >
```

The last technique is especially useful if your form uses more than one submit button for different actions. Refer to Chapter 17's section on `LookupDispatchAction` for details.

Equivalents

The JSF `<h:commandButton id="submit" type="SUBMIT">` allows you to emulate the functionality of the `<html:submit>` button. Of course, for you to use this with Struts, you need to embed it within the Strut-Faces `<s:form>` tag. Refer to Chapter 20 for details on `<h:commandButton>` and `<s:form>`.

text/password

`text` represents a one-line text input field. `password` represents a password field. The only difference between the two is that with `<html:password>`, the character keyed in by the user is not displayed; only asterisks or some other placeholder are displayed.

Usage Restrictions

The `property` attribute is required. It must be a child tag of `<html:form>`.

Attributes

All common attribute sets (evt-attrs, acc-attrs, ren-attrs, struts-attrs, init-attr, and the new err-attrs) are accepted, and have their usual meanings.

Besides these, there are a few attributes specific to both `text` and `password`:

- `size`: The length of the displayed field. If omitted, the decision is left to the browser.
- `readonly`: If `true`, the text field may not be edited.
- `maxlength`: Maximum number of characters to accept. The default is to allow an unlimited number of characters.

The `password` tag has an additional boolean attribute called `redisplay`, which defaults to `true`. This causes the previously keyed password (when a form is redisplayed because of validation errors) to appear as asterisks. Unfortunately, the actual password is also embedded in the HTML for that page. You should set `redisplay=false` to prevent passwords being harvested from cached pages.

Examples

The code

```
<html:text  property="email" />
```

specifies a text input field for the property called `email` on the current form bean.

```
<html:password property="pwd" />
```

specifies a password field.

Equivalents

JSF's `<h:inputText>` is an equivalent for `<html:text>`, but you need to enclose it within the Struts-Faces `<s:form>` tag so that the associated form bean is populated with the submitted value:

```
<s:form ...
   <!-- equivalent to html:text -->
   <h:inputText  id="email" binding="#{MyFormBean.email}" />

   <!-- equivalent to html:password -->
   <h:inputSecret  id="pwd" binding="#{MyFormBean.pwd}" />
   ...
</s:form>
```

textarea

This tag represents a multiple-line text input field.

Usage Restrictions

The `property` attribute is required. It must be a child tag of `<html:form>`.

Attributes

All common attribute sets (evt-attrs, acc-attrs, ren-attrs, struts-attrs , init-attr, and the new err-attrs) are accepted, and have their usual meanings.

Besides these, there are a few attributes specific to `textarea`:

- `cols`/`rows`: The number of columns (`cols`) or rows to display. The browser makes the decision if either is omitted.
- `readonly`: If `true`, the text field may not be edited.

Examples

The code

```
<html:textarea  property="memo" cols="8" rows="3"/>
```

specifies a `textarea` input field for the property called `memo` on the current form bean.

Equivalents

JSF's `<h:inputTextarea>` is an equivalent for `<html:textarea>`, but you need to enclose it within the Struts-Faces `<s:form>` tag so that the associated form bean is populated with the submitted value:

```
<s:form ...
   <!-- equivalent to html:textarea -->
   <h:inputTextarea  id="memo" binding="#{MyFormBean.memo}"
                  cols="8" rows="3" />
   ...
</s:form>
```

xhtml

This tag causes the Struts tags on the current page to render themselves as XHTML 1.0. This is essentially an XML-conformant version of HTML. You should be aware that some older browsers may balk at XHTML elements like `<input />` instead of HTML's `<input>`.

Usage Restrictions

This tag isn't "inherited" if you use a JSP include. You must put this tag on each and every JSP and JSP fragment that you want rendered as XHTML.

Attributes

None.

Examples

`<html:xhtml/>` at the start of the page is all you need.

Equivalents

None.

The Bean Tag Library

The Bean tag library has tags for reading the properties of JavaBeans and for writing text. There are two reasons why you'd use the Bean tags instead of hard-coding text into your JSPs. The first is to enable internationalization, and the second is to avoid using scriptlets.

Table C-10 gives a synopsis of tags in the Bean library, based on Apache's documentation. Refer to `www.apache.org/licenses/LICENSE-2.0` for a copy of the Apache License.

Table C-10. *Synopsis of the Bean Tag Library*

Tag	Usage/Comments
`message`	Writes static text based on the given key.
`write`	Writes the value of the specified JavaBean property.
`cookie/header/parameter`	Each exposes a scripting variable based on the value of the specified `cookie/header/parameter/` variable.
`define`	Each exposes a scripting variable based on the value of the specified JavaBean.
`page`	Each exposes a scripting variable based on the value of the specified page-scoped variable.
`include`	Allows you to call an external JSP or global forward or URL and make the resulting response data available as a variable. The response of the called page is not written to the response stream.
`resource`	Allows you to read any file from the current webapp and expose it either as a `String` variable or an `InputStream`.
`size`	Defines a new JavaBean containing the number of elements in a specified `Collection/Map/array`.
`struts`	Exposes the specified Struts internal configuration object as a JavaBean.

Struts-EL Tags for the Bean Tag Library

Only the following tags in the Bean library have EL-enabled versions: `include`, `message`, `page`, `resource`, `size`, and `struts`. (Note: EL-enabled tags are those that allow you to use EL expressions. Refer to Chapter 10 for examples.)

cookie/header/parameter

Each exposes a scripting variable based on the value of the specified cookie/header/parameter.

Usage Restrictions

The `id` and `name` attributes are required.

Attributes

- `name`:The name of the cookie/header/parameter to retrieve.
- `id`: The name of the exposed variable. Scriptlets and other custom tags will be able to access properties of the cookie/header/parameter using this name.
- `value`: The default value to return in case the cookie/header/parameter of the given name can't be found.
- `multiple`: Specifies how multiple cookies/headers/parameters with the same name are to be handled. The exact value of this attribute is unimportant. If the `multiple` attribute is present, then the variable exposed by `id` is an array of the corresponding type (`Cookie[]`/`String[]`/`String[]`). If `multiple` is not present, then the first occurring cookie/header/parameter is bound to the exposed variable.

Examples

Consider the following URL:

```
http://www.kenyir.org/mypage.jsp?command=test&action=save&id=12345
```

Here's how to expose the parameter named `command` on the requested URL:

```
<bean:parameter name="command" id="cmd" value="" />
Command is: <%=cmd%>
```

If there were more than one parameter named `command`, for example:

```
http://www.kenyir.org/mypage.jsp?command=print&command=save
```

you'd access them all in this way:

```
<bean:parameter name="command" id="cmd" value="" multiple="true"/>
<logic:iterate name="cmd" id="aCommand">
   <bean:write name="aCommand"/>
</logic:iterate>
```

The other tags (`header` and `cookie`) have similar examples.

Equivalents

The JSTL `<c:set>` is a replacement for these tags.

Here's how to use `<c:set>` as an equivalent tag for the case where the `multiple` attribute is not specified. The exposed variable is called `myVar*` and the corresponding name of the cookie/header/parameter is always `myAttr`.

```
<c:set var="myVar1" value="${cookie.myAttr}" />
<c:set var="myVar2" value="${header.myAttr}" />
<c:set var="myVar3" value="${param.myAttr}" />
```

If the `multiple` attribute is specified, then you'd use one of the following instead:

```
<c:set var="myVar4" value="${pageContext.request.cookies.myAttr}" />
<c:set var="myVar5" value="${headerValues.myAttr}" />
<c:set var="myVar6" value="${paramValues.myAttr}" />
```

Note that there is also a `<c:remove>` to delete a declared variable.

define

This tag exposes a variable based on data from (1) a given `String`, or (2) another JavaBean. Scriptlets and other tags can access this exposed variable like any other. This might not seem useful, but it is. Refer to the examples for details.

Refer also to the entries for `cookie/header/parameter` and `page`, which are related tags.

Usage Restrictions

The `id` attribute is required as the exposed variable's name. You also need to either specify `name/property/scope`, or `value`, or nest the exposed variable's value in the body of the `<bean:define>` tag.

Also, note that you can define a variable only once. Attempting to define a new variable with the same name on the same page will result in an `Exception` being thrown.

Attributes

- `id`: The name of the exposed variable. Scriptlets and other custom tags will be able to access the new variable using this name.
- `name`/`property`/`scope`: These attributes are used to locate an object to expose, with the given `name` and optional `property` and `scope`. If `scope` isn't specified, all scopes are searched for the named object.
- `value`: The `String` value to expose. This binds the variable exposed using the `id` attribute to the `String` specified by the `value` attribute.
- `toScope`: The scope on which to place the new variable. Can be `page`, `request`, `session`, or `application`.
- `type`: The fully qualified classname for the variable exposed using the `id` attribute. If not specified, `java.lang.String` is used when you also specify the `value` attribute. It is also implied when you use the nested technique (see the examples). If you use the `name`/`property`/`scope` method, then the implicit classname is `java.lang.Object`.

Examples

Here are a few examples of how you'd expose a `String` or variable using `<bean:define>`:

```
<bean:define id="myVar1" name="myAttr"
       property="myProp" scope="session"/>

<bean:define id="myVar2" name="myAttr"  scope="session"/>
<bean:define id="myVar3" value="Hello World" toScope="request" />
<bean:define id="myVar4">Here's looking at you kid!</bean:define>

<bean:define id="myVar5">
   <!-- myAttr.getMyProp() must never return null -->
  <bean:write  name="myAttr" property="myProp"/>
</bean:define>
```

In the last example, if `myAttr.getMyProp()` returned `null`, then an exception would be thrown. Otherwise, `myVar5` is equal to the value of `myAttr.getMyProp()` at the time it was called.

You can use `<bean:define>` to help you localize validations (see Chapter 12) if you use it with Struts-EL tags. Recall that in Chapter 12's treatment of the subject of localizing validations, we explained that you could use the trick of embedding the localized format `String` along with the form data (refer to Listing 12-3). An `Action` was used to populate the

hidden field. A more elegant way populating the hidden field is to use `<bean:define>` and `<html-el:hidden>` like so:

```
<bean:define id="dateFormat">
   <bean:message key="myapp.formats.dateFormat"/>
</bean:define>
<html-el:hidden  property="${dateFormat}" />
```

The message pointed to by `myapp.formats.dateFormat` is the date format, and of course, this is automatically localized by Struts. No `Action` subclass needed!

Equivalents

The JSTL `<c:set>` can be used to replace `<bean:define>`. The previous examples ported to `<c:set>` would be

```
<c:set var="myVar1" value="${sessionScope.myAttr.myProp}"/>
<c:set var="myVar2" value="${sessionScope.myAttr}"/>
<c:set var="myVar3" value="Hello World" scope="request" />
<c:set var="myVar4">Here's looking at you kid!</c:set>

<c:set var="myVar5">
  <!-- myAttr.getMyProp() must never return null -->
  <bean:write name="myAttr" property="myProp"/>
</c:set>
```

The last example is similar:

```
<c:set var="dateFormat">
   <bean:message key="myapp.formats.dateFormat"/>
</c:set>
<html-el:hidden  property="${dateFormat}" />
```

include

This interesting tag allows you to call an external JSP or global forward or URL and make the resulting response data available as a variable. The response of the called page is not written to the response stream.

The received output is HTML encoded (< is replaced with `<`, etc.) if the exposed variable is displayed with `<bean:write>`.

Usage Restrictions

You must specify the `id` attribute and either `forward`, `page`, or `href`.

Attributes

- `id`: The name of the exposed variable. Scriptlets and other custom tags will be able to access the new variable using this name. The resulting variable is page scoped.
- `forward`: The name of the global forward to call.
- `page`: The name of the JSP page on the current webapp to call. This name is module relative, and needs a slash prefix.
- `href`: The absolute URL of the external page—for example: `http://www.kenyir.org/code/index.html`.
- `anchor`: The name of the HTML anchor on the called page.
- `transaction`: If set to `true`, includes the transaction token (see the entry for `<logic:redirect>` for an explanation) in the response.

Examples

This example reads the given external page and then displays the actual contents. Remember, any received HTML is encoded:

```
<bean:include id="myPage" href="http://localhost:8080/lilldep/"/>
<bean:write name="myPage" />
```

Equivalents

The closest equivalent is JSTL's `<c:import>`, which calls a URL and exposes the received data as either a `String` or a `Reader`.

The previous example can easily be translated using JSTL:

```
<c:import  var="myPage" url="http://localhost:8080/lilldep/" />
<c:out value="${myPage}"/>
```

Note that you can also perform parameter replacement on the received data using nested `<c:param>` tags.

message

This tag displays an internationalized message. You may specify up to five replacement parameters.

Usage Restrictions

You must specify either the `key` attribute or the `name/property/scope` combination.

Attributes

- `key`: The key of the message resource to display. If the `locale/bundle` combination isn't specified, then the default message resource file and user's current locale are used to determine which message resource to display.

- `name/property/scope`: These attributes are used to derive a key, based on an object with the given name and optional property and scope. If `scope` isn't specified, all scopes are searched for the named object.

- `locale/bundle`: These attributes are used to specify a different `Locale` object or message resource file. `locale` specifies a key that can be used to look up the `Locale` object stored on the current session. The `bundle` attribute is explained in more detail in the following subsection.

- `arg0/arg1/arg2/arg3/arg4`: These are the values of the first, second, third, fourth, and fifth replacement parameters.

Using Multiple Message Resource Files

Struts allows you to declare more than one message resource file (that is, more than one `Application.properties` file, for the same locale). Each such message resource file is given a unique name, referred to in the Struts tags using the `bundle` attribute. The default message resource file has no name, and it is the one that is used if the `bundle` attribute isn't specified.

You should be aware that there are problems with using multiple message resource files:

- There are reported problems getting it to work with the Validator framework.

- It makes maintenance more difficult. This is because in addition to specifying the key, you'd have to indicate the bundle name (if it's not the default one). If you later decide to move a message from one file to another (common if you're consolidating an app to ensure uniformity of displayed messages or prompts), then you'd have to change the bundle value of every tag that used that particular message.

So you should try to avoid using this feature if possible.

An alternative is to use a naming convention for keys, like the dot naming convention (e.g., `app.error.prompt.login`) I've used for message keys throughout this book. This lets you to create separate namespaces, and so obviates the need for multiple message resource files.

Now that you're aware of the dangers, here's how you'd declare multiple message resource files in your `struts-config.xml` file:

```
<!-- the default message resource file:   -->
<message-resources   parameter="Application" />

<!-- another message resource file -->
<message-resources   parameter="Application2"
                          key="myOtherResourceFile"/>
```

In this example, the second message resource file is `Application2.properties`, and is stored in the same place as the default `Application.properties` file (that is, in `/WEB-INF/classes/`).

To display a prompt from the default file, you'd use

```
<bean:message key="app.error.prompt.login" />
```

To display a prompt from the second file, you'd use

```
<bean:message bundle="myOtherResourceFile"
                 key="app.prompt.checkdata" />
```

The extra `bundle` attribute tells Struts which message resource file to use.

Examples

Here's how to display a message:

```
<bean:message key="app.prompt.login" />
```

Here's how to use a JavaBean whose `toString()` results in a key:

```
<bean:message name="myLoginMessageKey" />
```

Here's how to use a JavaBean whose property gives a key:

```
<bean:message name="MyBean" property="formatKey" />
```

This example calls `MyBean.getFormatKey()` and uses it as the message key. `toString()` is called on the return value if necessary.

If the message has replacement parameters, for example:

```
app.prompt.logoff=Logoff {0}
```

you can use the following to perform the replacement (using EL):

```
<bean:message-el  key="app.prompt.logoff"  arg0="${user.name}" />
```

Equivalents

There are equivalents in both JSTL and JSF. With JSTL, you'd use the Formatting library:

```
<fmt:bundle basename="Application">
  <fmt:message  key="app.prompt.logoff">
    <fmt:param value="${user.name}"/>
  </fmt:message>
</fmt: bundle>
```

This snippet is an equivalent for the last example, with a single replacement parameter.

If you're using JSF, you might find it convenient to use the Struts-Faces library (see Chapter 20) to expose the default message resource file:

```
<s:loadMessages var="messages"/>
<h:outputFormat value="#{messages['app.prompt.logoff']}">
   <f:param value="#{user.name}"/>
</h:outputFormat>
```

This is the equivalent of the previous JSTL example.

Lastly, there is `<s:message>` from the Struts-Faces integration library.

page

This handy tag exposes objects from the page context as variables that may be accessed with scriptlets or custom tags.

The `PageContext` object is the central repository of all state for the current page. This includes the session (`HttpSession`) and request (`HttpServletRequest`) that were described in Chapter 2. There is more:

- `application`: The `javax.servlet.ServletContext` object, obtained by calling `pageContext.getServletContext()`.
- `config`: The `javax.servlet.ServletConfig` object, obtained by calling `pageContext.getServletConfig()`. Don't confuse this with Struts' internal configuration objects.
- `response`: The `HttpServletResponse` object associated with the page.

You should consult a recent reference on servlets (or Google the classname to get the JavaDoc) to find out more about these objects.

Usage Restrictions

Both `id` and `property` attributes are required.

Attributes

- `id`: The name of the exposed variable. Scriptlets and other custom tags will be able to access the new variable using this name. This attribute is required.
- `property`: The value is `application`, `session`, `request`, `config`, or `response`; used to get the associated objects from the page context. This attribute is required.

Examples

The snippet

```
<bean:page id="myVar" property="session" />
```

exposes the session object (`HttpSession`) associated with the enclosing page as a bean named `myVar`. The other page context objects are similarly exposed.

Equivalents

JSTL's `<c:set>` can be used to replace `<bean:page>`. Let's look at some examples.

To expose the application, use this:

```
<c:set var="myVar" value="${pageContext.servletContext}"/>
```

To expose the session, use this:

```
<c:set var="myVar" value="${pageContext.session}"/>
```

To expose the request, use this:

```
<c:set var="myVar" value="${pageContext.request}"/>
```

To expose the `config` object, use this:

```
<c:set var="myVar" value="${pageContext.servletConfig}"/>
```

To expose the response, use this:

```
<c:set var="myVar" value="${pageContext.response}"/>
```

You should note that the JSTL implicit object `sessionScope` is not the same as `pageContext.session`. The former is a `Map` containing key/value pairs, and the latter is the actual `HttpSession` object, whose properties you can read.

resource

`resource` allows you to read any file from the current webapp and expose it either as a `String` variable or an `InputStream`. See also `<bean:include>`.

Usage Restrictions

The `id` and `name` attributes are required.

Attributes

- `id`: The name of the exposed variable. Scriptlets and other custom tags will be able to access the new variable using this name. This attribute is required.
- `name`: The module-relative name of the resource to load.
- `input`: If this attribute is present (the actual value is irrelevant), then the exposed variable is of type `InputStream`; otherwise it's a `String`.

Examples

Frankly, it's difficult to see how this tag can be useful, unless you're using some other custom tags that can read and parse the `InputStream`. Here's an example using a hypothetical RSS source (`/myRss.xml`, which contains RSS markup) and a hypothetical `<rss>` custom tag to parse and display the output:

```
<!-- expose the RSS XML data: -->
<bean:resource  id="rssSrc"  name="/myRss.xml"  input="yes" />

<!-- display the RSS data as HTML: -->
<rss:write  name="rssSrc" />
```

Equivalents

The closest equivalent is JSTL's `<c:import>`, which calls a URL and exposes the received data as either a `String` or a `Reader`. The previous example can easily be translated using JSTL:

```
<c:import  varReader="rssSrc" url="myRss.xml" />
<rss:write value="${rssSrc}"/>
```

size

This tag exposes the size of a given `Collection` or `Map` or array as a variable (of type `Integer`) that may be accessed with scriptlets or custom tags.

Usage Restrictions

The `id` attribute is required, and you must specify the array/`Collection`/`Map` whose size you want measured, using either the `collection` attribute or a `name`/`property`/`scope` combination.

Attributes

- `id`: The name of the exposed variable. Scriptlets and other custom tags will be able to access the new variable using this name. This attribute is required.
- `name`/`property`/`scope`: These attributes are used to locate the `Collection`/`Map`/array, with the given name and optional property and scope. If `scope` isn't specified, all scopes are searched for the named object.
- `collection`: This is a very handy alternative to the `name`/`property`/`scope` combination. You use a scriptlet to calculate the value of this attribute, for example:

```
<bean:size id="mySize"
   collection="<%=MyObject.calculateCollection() %>" />
```

- The rationale behind this attribute is discussed in the entry for `<logic:iterate>`.

Examples

Here's a simple example that exposes the size of a given collection:

```
<bean:size id="mySize" name="MyCollection"/>
<bean:write name="mySize"/>
```

Equivalents

You can use JSTL's `<c:set>` in conjunction with the JSTL `length` function:

```
<c:set var="myVar" value="${fn:length(MyCollection)}" />
```

Remember to declare the JSTL function tag library.

struts

This tag exposes a Struts global forward, or form bean or form handler (the `ActionMapping` for the form handler) as a variable, accessible to scriptlets and other tags. This can be useful if you want JSTL to access Struts' internal variables.

Usage Restrictions

The `id` attribute is required. You must also specify the `forward`, `formBean`, or `mapping` attribute.

Attributes

- `id`: The name of the exposed variable. Scriptlets and other custom tags will be able to access the new variable using this name. This attribute is required.
- `forward`: The name of the global forward to expose.
- `formBean`: The name of the declared form bean to expose.
- `mapping`: The name of the form handler to expose.

Examples

Consider this `struts-config.xml` snippet:

```
<form-beans>
   <form-bean name="RegistrationFormBean"
              type="net.thinksquared.reg.RegistrationFormBean" />
</form-beans>

<global-forwards>
   <forward name="myForward"  path="/index.jsp" />
</global-forwards>

<action-mappings>
   <action path="/Login"
           type="net.thinksquared.reg.RegistrationAction"
       validate="true"
           name="RegistrationFormBean"
          input="myInput.jsp">

           <forward name="success"  path="registered.jsp" />
           <forward name="new-user" path="new-user.jsp" />
   </action>
</action-mappings>
```

Here's how you would expose the global forward:

```
<bean:struts id="fwd" forward="myForward"/>
<bean:write name="fwd"/>
<bean:write name="fwd" property="path" />
```

and the form bean:

```
<bean:struts id="fbean" formBean="RegistrationFormBean"/>
<bean:write name="fbean"/>
<bean:write name="fbean" property="type" />
```

and the form handler:

```
<bean:struts id="fhandler" mapping="/Login"/>
<bean:write name="fhandler"/>
<bean:write name="fhandler" property="input" />
```

Equivalents

None.

write

This tag writes the specified bean property to the response stream. It allows for some formatting of the given bean property.

Usage Restrictions

The `name` attribute is required.

Attributes

- `name`/`property`/`scope`: These attributes are used to locate the JavaBean with the given name and optional property and scope. If `scope` isn't specified, all scopes are searched for the named object.

- `ignore`: If `true`, silently fails if the named JavaBean is not found. The default value (`false`) causes an exception to be thrown.

The remaining attributes are used to format the bean's property:

- `filter`: If set to `true` causes the property's `String` value to be HTML encoded. For example, the ampersand (`&`) would be encoded as `&` and > would be encoded as `>`. The default is `true`.

- `format`/`formatKey`: Both specify a format string that is used to format the bean's property. `format` uses its value literally as a format string. `formatKey` uses its value as a key to look up the actual format string stored in the message resources file. Unless the `locale`/`bundle` combination is specified, the current user's locale and the default message resource bundle is used.

- `locale/bundle`: These attributes are used to specify a different `Locale` object or message resource file. `locale` specifies a key that can be used to look up the `Locale` object stored on the current session. The `bundle` attribute is explained in more detail in the entry for `<bean:message>`.You'd use these in conjunction with `formatKey` to specify a locale different from the current user's locale or a message resource file different from the default one.

Examples

The code

```
<bean:write name="myString" />
```

will display the `String` referred to by `myString`.

```
<bean:write name="MyBean" property="message />
```

will display the value of `MyBean.getMessage()`.

```
<bean:write name="myFloat" format="#.##" />
```

will display `myFloat = 3.1416` as `3.14`.

```
<bean:write name="myCalendar" format="dd/mm/yyyy" />
```

will format the date represented by `myCalendar` as `dd/mm/yyyy`. For example, `20 Feb 2006` will be displayed as `20/02/2006`.

```
<bean:write name="myCalendar" formatKey="app.format.dateFormat" />
```

will use the value of the message key `app.format.dateFormat` as a format string.

Equivalents

You can use JSTL's `<c:out>` to display a bean's properties:

```
<c:out value="${MyBean.message}"/>
```

displays the value of `MyBean.getMessage()`. To format the output, you'd use JSTL's Formatting tag library:

```
<fmt:formatNumber  value="${myFloat}" pattern="#.##" />
```

as an equivalent to the number formatting given earlier, or

```
<fmt:formatDate  value="${myCalendar}" pattern="dd/mm/yyyy" />
```

JSTL's Formatting tag library has numerous variations on formatting strings, dates, currencies, and numbers. You should consult the latest specification if you're interested in learning more.

Lastly, the `<s:write>` tag from the Struts-Faces integration library is yet another equivalent.

The Logic Tag Library

The Logic tag library provides tags for conditional processing, looping, and flow control.

- Conditional processing: `equal/notEqual`, `empty/notEmpty`, `greaterEqual/lessEqual/greaterThan/lessThan`, `match/notMatch`, `messagesPresent/messagesNotPresent` and `present/notPresent`.
- Looping: The `iterate` tag.
- Flow control: The `forward` and `redirect` tags.

All of these tags, except the ones that involve flow control, have JSTL equivalents, so you should use the JSTL versions if you can.

Common Attribute Sets

There are two common attribute sets for the Logic tag library. The first, which we'll call the Base Object attribute set (or **base-attrs** for short) is illustrated in Table C-11, and pertains to the JavaBean object that the corresponding tags apply to.

Table C-11. *The base-attrs Attribute Set*

Attribute Name	Usage
`name`	The name of the base object. If this is omitted, then the implicit base object (if any) is used. If the tag is enclosed in an `<html:form>`, the implicit base object is the form bean associated with the form.
`property`	The property on the base object, whose value is used in the functioning of the tag. For example, if the tag were a comparison tag, then the value of `property` on the base object is used in the comparison. This assumes a corresponding `getXXX()` function on the base object.
`scope`	The scope on which to search for the object given by the `name` property. If `scope` is undeclared, then all scopes are searched for the object specified by the `name` property. The valid scopes are `page`, `request`, `session`, and `application`.

Some tags allow you to use values from other sources (that is, other than the property on the base object) in order to do their job. These Extended Property attributes (**exprop-attrs**) are listed in Table C-12.

Table C-12. *The exprop-attrs Atribute Set*

Attribute Name	Usage
`cookie`	The name of the cookie to use.
`header`	The name of the HTTP request header variable to use. The name match is case insensitive.
`parameter`	The name of the URL request parameter to use. If there are more than one, the first one that occurs is used.

Selector Attributes

Many tags in the Logic library use selector attributes. These attributes are used to select the object or property with which to perform that tag's function. The selector attributes are `cookie`, `header`, `name`/`property`, and `parameter`.

Struts interrogates these attributes in this sequence. For example, if you specify both `cookie` and `parameter`, only the `cookie` attribute is used to perform the tag's task.

Note that the `name`/`property` attributes are considered a pair in this respect. That is, if you specify the name, you may optionally specify the property as well. You cannot specify just the property alone—you must specify the name as well, if they function as a selector attribute. The reference for each tag will tell you if this is the case or not.

Struts-EL Tags for the Logic Tag Library

Only the following tags in the Logic library have EL-enabled versions: `forward`, `iterate`, `match`/`notMatch`, `messagesPresent`/`messagesNotPresent`, `present`/`notPresent`, and `redirect`. Other tags have simpler JSTL equivalents.

■ **Note** EL-enabled tags are those that allow you to use EL expressions. Refer to Chapter 10 for examples.

---

empty/notEmpty

`empty` conditionally executes its body if the specified property (or base object) is `null`, a zero-length `String`, or an empty `Collection` or `Map`. If the `property` attribute isn't specified, then the test is run against the base object given by the `name` attribute.

`notEmpty` is the converse of `empty`.

Usage Restrictions

The `name` attribute must be specified, even if nested within an `<html:form>`.

Attributes

Only the base-attrs attribute set is applicable.

Examples

The following example shows the use of the `<logic:empty>` tag to conditionally process the nested `<bean:write>` tag if `MyCollection` is empty:

```
<logic:empty name="MyCollection">
   <bean:write name="MyCollection" property = "myproperty[13]" />
</logic:empty>
```

The next example illustrates the use of the `property` tag:

```
<logic:empty name="contact" property="email">
   <bean:write name="contact" property="email"/>
</logic:empty>
```

The `<bean:write>` is executed only if the `getEmail()` function returns `null`, or a zero-length `String`. You can also use the `scope` variable to specify a scope under which to look for a bean:

```
<logic:notEmpty name="MyBean" scope="request">
   <bean:write name="MyBean" property = "myproperty" />
</logic:empty>
```

In the previous example, the `<bean:write>` is called on the base object if `MyBean` can be found on the request scope.

Equivalents

You can easily create equivalents for `empty` and `notEmpty` using JSTL's `<c:if>` or `<c:choose>...<c:when>`. For example using `<c:if>`, the previous examples would be

```
<c:if test="${empty MyCollection}">
   <bean:write name="MyCollection" property = "myproperty[13]" />
</c:if>
```

and

```
<c:if test="${empty contact.email}">
   <bean:write name="contact" property="email"/>
</c:if>
```

You'd have to use JSTL's implicit objects (`requestScope`, `sessionScope`, etc.) to specify a scope:

```
<c:if test="${ not empty requestScope.MyBean}">
   <bean:write name="MyBean" property = "myproperty" />
</c:if>
```

equal/notEqual

`equal` checks that the given property (or extended property) is equal to a given value. If so, the body of the equal tag is executed. `notEqual` is the converse of `equal`.

Usage Restrictions

The `value` attribute is required. You must specify one selector attribute (see the start of this section for a definition) as well.

Attributes

Both base-attrs and exprop-attrs are accepted. There is also an additional `value` attribute, which is a constant value that the property or extended property will be compared to.

Examples

The following example shows the use of the `<logic:equal>` tag to conditionally process the nested `<bean:write>` tag if the variable client is equal to the `String` "Joe":

```
<logic:equal name="client" value="Joe">
   <bean:write name="client" />
</logic:equal>
```

The next example illustrates the use of the `parameter` attribute:

```
<logic:equal parameter="username" value="Susan">
   Hello Susan!
</logic:equal>
```

The body is executed only if there is a parameter named `username` on the request URL with the value `Susan`. If there were more than one such parameter, the first one that occurs on the URL is used.

You can also use the `scope` variable to specify a scope under which to look for a bean:

```
<logic:notEqual name="MyBean" property="email"
               scope="session" value="joey@joey.com">
   <bean:write name="MyBean" property = "email" />
</logic:notEqual>
```

In this example, the <bean:write> is called on the base object if there is a bean on the session scope called MyBean, and its getEmail() function returns a value equal to joey@joey.com. Note that if MyBean is null, or does not have a getEmail() function, an exception is thrown.

Equivalents

You can easily create equivalents for equal and notEqual using JSTL's <c:if> or <c:choose>...<c:when>. For example using <c:if>, the previous examples would be

```
<c:if test="${client == 'Joe'}">
   <bean:write name="client" />
</c:if>
```

The next example illustrates the use of the parameter attribute:

```
<c:if test="${param.username == 'Susan'}">
   Hello Susan!
</c:if>
```

You can also use the scope variable to specify a scope under which to look for a bean:

```
<c:if test="${sessionScope.MyBean.email != 'joey@joey.com'}">
   <bean:write name="MyBean" property = "email" />
</c:if>
```

Also note that unlike the Struts tags that only admit a constant value, the JSTL equivalents have no such restriction. Here's an example of comparing a variable against another (this is impossible to implement with the Struts tags because value must be constant—unless of course, you're willing to use scriptlets):

```
<c:if test="${cookie.userlogin != MyBean.userlogin}">
   <bean:message key = "error.myapp.login" />
</c:if>
```

forward

This tag causes the page to be forwarded to the specified global forward (see Chapters 9 and 17). This tag has no body. See also <logic:redirect>.

Usage Restrictions

The name attribute is required.

Attributes

The name attribute is the name of the global forward to use.

■ **Note** If you wish to forward to a Tiles definition, a hack is to do this from an `Action` subclass, which is in turn exposed as a global forward.

Examples

Consider the global forwards declared in `struts-config.xml`:

```
<global-forwards>
   <forward   name="login"   path="/login.jsp"/>
   <forward   name="logoff"   path="/Logoff.do"/>
</global-forwards>
```

Then, the forward

```
<logic:forward name="login"/>
```

would cause the page `login.jsp` to be displayed. Similarly, the forward

```
<logic:forward name="logoff"/>
```

would cause the `Logoff.do` form handler to be called. You can use this to load a Tiles definition.

Equivalents

None.

greaterEqual/lessEqual/greaterThan/lessThan

`greaterEqual` is a conditional tag that executes its body if the given property (or extended property) is greater than or equal to the constant value given by the `value` attribute. The other tags in this group are similarly defined.

Note that if value is not a numeric value, it is interpreted as a `String`, and a comparison between `Strings` (using Java's standard comparisons between `Strings`) will be performed.

Usage Restrictions

The `value` attribute is required. You must specify one selector attribute (see the start of this section for a definition) as well.

Attributes

Both base-attrs and exprop-attrs are accepted. There is also an additional `value` attribute, which is a constant value that the property or extended property will be compared to.

Examples

The example that follows shows the use of the <logic:greaterThan> tag to conditionally process the nested tags (not shown) if the variable hits is greater or equal to 10:

```
<logic:greaterEqual name="hits" value="10">
   //other tags here.
</logic:greaterEqual>
```

The next example illustrates the use of the cookie attribute:

```
<logic:greaterThan cookie="logintries" value="3">
   <bean:message key= "error.myapp.login.tries-exceeded" />
</logic:greaterThan>
```

The <bean:message> is executed only if the cookie called logintries is greater than 3.

Equivalents

You can easily create equivalents for these tags using JSTL's <c:if> or <c:choose>...<c:when>. For example using <c:if>, the previous examples would be

```
<c:if test="${hits >= 10}">
   //other tags here.
</c:if>
```

The next example illustrates the use of the cookie attribute:

```
<c:if test="${cookie.logintries > 3}">
   //other tags here.
</c:if>
```

Also note that unlike the Struts tags that only admit a constant value, the JSTL equivalents have no such restriction.

iterate

The <logic:iterate> tag can be used to iterate over Collections, Maps, Enumerations, arrays, or Iterators (referred to as the "iteratable object"). The tags (or static text) nested within an <logic:iterate> are executed into the JSP's response stream (i.e., written to the final HTML page).

Usage Restrictions

The id attribute is required. You must also specify the iteratable object (using either the name/property attributes or the collection attribute).

Attributes

The `iterate` tag uses base-attrs to specify the base object that represents the `Collection`/`Map`/`Enumeration`/array/`Iterator` to iterate over. Besides these, several other attributes can be used:

- `collection`: The `name`/`property` combination only allows you to loop over iteratable objects that may be read as JavaBean properties. This can be quite restrictive. For example:

  ```
  <logic:iterate name="MyBean" property="myCollection" ...
  ```

  implicitly calls the function

  ```
  MyBean.getMyCollection()
  ```

- in order to determine the iteratable object. Many Java classes however, do not use the JavaBeans `getXXX()` convention, so a different approach is needed for them. The solution is to use the `collection` attribute in combination with a scriptlet:

  ```
  <logic:iterate collection="<%=MyObject.iterator()%>" ...
  ```

- `id`: This is the name given to a single element read from the iteratable object.

- `indexId`: The current index; starts from zero.

- `length`: The maximum number of elements to read from the iteratable object. If not present, all elements are read.

- `offset`: The index of the first element to return from the iteratable object. For example, if the iteratable object is an array `[a, b, c]`, using `offset="1"` makes the loop read `b` and `c` only. Note that the offset is zero based, as it is in Java.

- `type`: The fully qualified classname of the elements read from the iteratable object. If an element doesn't match this classname, a `ClassCastException` is thrown.

Examples

Here's the basic usage:

```
<logic:iterate name="MyBean" property="myCollection" id="anElement">
   <!-- anElement.toString() called -->
   <bean:write name="anElement" />
</logic:iterate>
```

and with the collection attribute:

```
<logic:iterate collection="<%=MyObject.iterator()%>" id="anElement">
   <!-- anElement.toString() called -->
   <bean:write name="anElement" />
</logic:iterate>
```

And using the indexId attribute to display the current iteration number:

```
<logic:iterate name="MyContacts" id="aContact" indexId="count">
   <bean:write name="count" />
   <bean:write name="aContact" property="email"/>
</logic:iterate>
```

Equivalents

JSTL's <c:forEach> is a better replacement for <logic:iterate>. Here are the previous examples using JSTL:

```
<c:forEach var="anElement" items="${MyBean.myCollection}">
   <c:out value="${anElement}"/>
</c:forEach>
```

There is no "collection" attribute needed for the next example:

```
<c:forEach var="anElement" items="<%=MyObject.iterator()%>">
   <c:out value="${anElement}"/>
</c:forEach>
```

Lastly, you expose the iteration number this way:

```
<c:forEach var="aContact" items="${MyContacts}" varStatus="count">
   <c:out value="${count}"/>
   <c:out value="${aContact.email}"/>
</c:forEach>
```

match/notMatch

match is a conditional tag that executes its body if the given property (or extended property) contains the value attribute as a substring. notMatch is the converse of match.

Usage Restrictions

The value attribute is required. You must specify one selector attribute (see the start of this section for a definition) as well.

Attributes

Both base-attrs and exprop-attrs are accepted, and have their usual meaning. Besides these, there are two additional attributes:

- `value`: A constant substring that the property or extended property will be matched for. For example, if the property or extended property results in the string "Visit Zimbabwe", then `value="im"` would cause a match, while `value="Zic"` would not.

- `location`: The value is either `start` or `end`, and specifies where the matching should take place within the source string. If omitted, a match anywhere will succeed.

Examples

Here's a simple example that looks for an email address with an `.sg` suffix:

```
<logic:match name="MyBean" property="email"
             value=".sg"    location="end">
   I see you're from Singapore!
</logic:match>
```

Or you could interrogate the HTTP request header:

```
<logic:notMatch  header="user-agent"  value="Windows">
   Try knoppix now: http://www.knoppix.net
</logic:notMatch>
```

Equivalents

You can easily create equivalents for these tags using JSTL's `<c:if>` (or `<c:choose>...<c:when>`) in conjunction with JSTL functions. (This is a standard JSTL library of functions. You must declare the JSTL function taglib in order to use it.) For example, using `<c:if>` the previous examples would be

```
<c:if test="${fn:endsWith(MyBean.email,'.sg')}">
   I see you're from Singapore!
</c:if>
```

and

```
<c:if test="${fn:contains(header.user-agent,'Windows')}">
   Try knoppix now: http://www.knoppix.net
</c:if>
```

messagesPresent/messagesNotPresent

`messagesPresent` is a conditional tag that executes its body if the given message or error message exists on the current request. `messagesNotPresent` is the converse of `messagesPresent`.

Note the following terminology:

- "Error messages" are `ActionMessage` instances stored in an `ActionMessages`/`ActionErrors` instance, which in turn is stored under the key `Globals.ERROR_KEY` in the request.

- "Messages" are `ActionMessage` instances stored in an `ActionMessages`/`ActionErrors` instance, which in turn is stored under the key `Globals.MESSAGE_KEY` in the request.

For example, your `ActionForm`'s `validate()` might return an `ActionErrors` instance that contains one or more `ActionMessage` instances. These `ActionMessage` instances are stored by key, as you've seen in Chapters 6 and 7. The key is called the "name" of the corresponding error message/message.

Both error messages and messages may be displayed with `<html:errors>`, or iterated with `<html:messages>`.

Usage Restrictions

None.

Attributes

There are three attributes: `name`, `property`, and `message`.

Note that the `name` and `property` attributes don't have their usual meaning. Instead, the following checks are made depending on the settings of these attributes:

- If no attributes are specified, `messagesPresent` simply checks if there any error messages in the current request. It executes its body if there are error messages. Setting `message="true"` causes `messagesPresent` to check for any messages instead.

- If the `property` attribute is specified, `messagesPresent` checks if there are any error messages with that name. Again, setting `message="true"` causes a check for the appropriate message instead.

- The `name` attribute allows you to look for error messages/messages on the request, under a key other than `Globals.ERROR_KEY` or `Globals.MESSAGE_KEY`. Note that if you specify the `message` attribute, the `name` attribute is ignored.

- If the `name` attribute is specified, then even a `String` or `String` array in the request will result in a match.

The various combinations are probably best illustrated with examples.

Examples

The simplest example tests for the presence of error messages:

```
<logic:messagesPresent >
   There are errors on this page!
</logic:messagesPresent>
```

Or you could test for a specific error message:

```
<logic:messagesPresent property="email">
   The email address field has an error!
</logic:messagesPresent>
```

If your `Action` posts messages, you can test for these as well:

```
<logic:messagesNotPresent property="alerts" messages=true ">
   The are no alerts for you.
</logic:messagesNotPresent>
```

And you can look for error messages or messages or `String`s or `String` arrays under a specific key on the request:

```
<logic:messagesPresent  name="mywebapp_debug_messages">
   The debug message is: <bean:write name="mywebapp_debug_messages"/>
</logic:messagesPresent>
```

Equivalents

None. There are, of course, hacks using scriptlets in combination with JSTL's `<c:if>`.

present/notPresent

`present` checks the request for the presence of an object under the specified property or extended property. If the object is present, the body of the tag is executed. `notPresent` is the converse of `present`.

Usage Restrictions

You must specify a selector attribute (see the start of this section for a definition) or user or role.

Attributes

Both base-attrs and exprop-attrs are accepted, and have their usual interpretation. If the `property` attribute is specified, `present` checks that the return value is not `null`.

In addition to these, there are other user and role attributes. These depend on the underlying servlet technology's facility to authenticate a user, and specify one or more roles for that user. How this is done is outside the scope of this book, so if you're interested in learning more, you should consult a recent reference on servlets. The `user` attribute specifies the name of the remote user for the current request. The `role` attribute is a comma-delimited string of roles. `present` evaluates its body if any one of these roles is a valid role of the current user.

Examples

The simplest usage checks for the presence of an object on the request:

```
<logic:present  name="myobject">
   My object is present!
</logic:present>
```

Similarly, the use of the `property` attribute ensures that the given property is not `null`:

```
<logic:present  name="contact" property="designation">
   The contact's designation is:
   <bean:write name="contact" property="designation"/>
</logic:present>
```

You can also check the current authenticated username:

```
<logic:present  user="donquixote">
   Hi Don!
</logic:present>
```

or that the current authenticated user has any of the specified roles:

```
<logic:notPresent  role="admin,staff">
  Sorry, you are not allowed to view this page!
</logic:notPresent>
```

Equivalents

You can easily create equivalents for these tags using JSTL's `<c:if>` or `<c:choose>...<c:when>`. For example using `<c:if>`, the previous examples would be

```
<c:if test="${myobject != null}">
      My object is present!
</c:if>
```

```
<c:if  test="${contact != null && contact.designation != null}">
   The contact's designation is:
   <c:out value="${contact.designation}"/>
</c:if >
```

```
<c:if  test="${pageContext.request.remoteUser == 'donquixote'}">
   Hi Don!
</c:if >
```

The last example—checking roles—unfortunately has no JSTL equivalent. However, there is a JSF extension for security that handles this case neatly. Check out `http://jsf-security.sourceforge.net` if you're interested.

redirect

This tag performs a page redirect. The tag has no body. See also `<logic:forward>`.

Usage Restrictions

You must specify either `forward`, `href`, `action`, or `page` to specify the base URL to redirect to.

Attributes

There are two main sets of attributes. The first set specifies the base URL of the page to redirect to. You have four ways to do this:

- `forward`: The name of the global forward to redirect to (e.g., `forward="login"`).
- `href`: The raw URL to redirect to (e.g., `href="http://www.kenyir.org"`).
- `action`: The name of the form handler (that is, action mapping) to redirect to. This must begin with a slash because it's relative to the default module (e.g., `action="/Logoff"`). (See Chapter 17 for more on modules.)
- `page`: The path to the web page on the current webapp to redirect to. This path is relative to the webapp's base folder. The path must begin with a slash (e.g., `page="/index.jsp"`).

`redirect` also allows you to specify parameters on the redirected URL:

- `anchor`: Lets you specify the name of an anchor. Anchors are standard HTML. They are used to label sections of a long web page for easy navigation.
- `name`/`property`/`scope`: These attributes are used to locate a `Map` instance on the request or session. The key of the map is the parameter's name, and the value is the corresponding parameter's value. In the case that the value is an array, multiple parameters are written to the URL, each with the same name but different values. You use a combination of these three variables if you want to create more than one URL parameter.

- `paramName`,`paramProperty`,`paramScope` and `paramId`: The first set of three attributes is used to locate a single object on the current request or session. If this object is not an array, its `toString()` is called and this is the single parameter value. If the object is an array, then you're creating multiple parameters with the same name but different values. The name of the parameter is given by `paramId`.

The remaining attributes are `useLocalEncoding` and `transaction`. `useLocalEncoding`, if set to `true`, tells Struts to use whatever the character encoding is for the current `HttpServletResponse`. The `transaction` attribute requires some elaboration.

Transaction Tokens

Struts has a mechanism to prevent a form from being submitted more than once. This might happen if the user clicks the form's submit button twice in quick succession. Ordinarily, Struts would treat this as two separate submissions. This, of course, is an error. A standard solution to this problem is to use a *transaction token*.

You have to use two `Action` subclasses. The first `Action` subclass is activated when the form is requested. It places a unique string on both the request and the session. This unique string is the transaction token. `Action` has the function

```
saveToken(HttpServletRequest request)
```

to do this for you. This function automatically generates the transaction token and places it on the request and session under appropriate keys. A simple example of the `execute()` function of this `Action` subclass is as follows:

```
public ActionForward execute(...){
   //create the transaction token
   saveToken(request);
   //give the user the form to fill
   return mapping.findForward("success");
}
```

The second `Action` subclass is the one that processes the form data once it is submitted by the user. This is what its `execute()` would look like:

```
public ActionForward execute(...){
   if(isTokenValid(request,true)){
      return mapping.findForward("duplicate-submission");
   }
   //process form data as usual.
   ...
}
```

The function `isTokenValid(request,true)` checks to ensure that the transaction token on the request matches that on the session. Once the check is done, it destroys the copy on the session. This means that if the user clicked the submit button a second time, there would be no copy on the session, causing `isTokenValid()` to return `false`.

However, you should be aware that the simple solution outlined in the preceding section has one big failing: the error page (`duplicate-submission`) sent back to the user might make him think that the form data wasn't been processed at all! Careful wording of the error message might help, but you still can't get the true `success` page to the user. More sophisticated solutions would solve such user interaction issues as well.

Coming back to the transaction attribute, if it's set to `true` then the value of the transaction token on the request is sent as a URL parameter.

Examples

Here is a simple `redirect` using the name of a global forward:

```
<logic:redirect forward="login" />
```

Next is an example of adding a single URL parameter. The name of the parameter (`paramId`) is `email-address`. The value of the parameter is determined by calling `contact.getEmail()`:

```
<logic:redirect action="/MyFormHandler" paramId="email-address"
             paramName="contact"  paramProperty="email" />
```

Here's how you would add multiple URL parameters, based on a `Map`. The `Map` is located by calling `contact.getAttributes()`, which is located on the session scope:

```
<logic:redirect action="/mypage.jsp" name="contact"
         property="attributes"      scope="session" />
```

Equivalents

None. No single tag in JSTL or JSF perform all the functions of `<logic:redirect>`.

The Nested Tag Library

The nested tag library allows you to apply tags relative to an object. You can emulate exactly the same functionality using nested properties (see Chapter 10 for a detailed exposition).

Unfortunately, to a programmer, the Nested taglib might feel like a kludge. For instance, suppose you wanted to write the contents of

```
MyBean.getMyPropertyA().getMyPropertyB()
```

You might expect to be able to do this:

```
<nested:root name="MyBean">
   <nested:nest property="myPropertyA">
      <bean:write property=”myPropertyB”/> //WRONG!!
   </nested:nest>
</nested:root>
```

Instead, the Nested tag library provides replacement tags for `<bean:write>` and many other tags. So, the correct code to perform the task is

```
<nested:root name="MyBean">
   <nested:nest property="myPropertyA">
      <nested:write property=”myPropertyB”/>
   </nested:nest>
</nested:root>
```

The replacement for `<bean:write>` is `<nested:write>`. Most of the Nested tag library consists of replacement tags. These are exact duplicates of their counterparts in other tag libraries:

- **Replacements for HTML**: `checkbox`, `errors`, `file`, `form`, `hidden`, `image`, `img`, `link`, `messages`, `multibox`, `options`, `optionsCollection`, `password`, `radio`, `select`, `submit`, `text`, and `textarea`
- **Replacements for Bean**: `define`, `message`, `size`, and `write`
- **Replacements for Logic**: `empty`, `equal`, `greaterEqual`, `greaterThan`, `iterate`, `lessEqual`, `lessThan`, `match`, `messagesNotPresent`, `messagesPresent`, `notEmpty`, `notEqual`, `notMatch`, `notPresent`, and `present`

Since these tags are exact duplicates of the originals, I won’t repeat their functions here. You should consult the original tag’s reference in this appendix. Excluding these replacement tags, the Nested tag library has just three tags of its own that require some explanation. The section that follows tackles this task.

Struts-EL Tags for the Nested Tag Library

There are no Struts-EL tags for the Nested tag library.

■ **Note** EL-enabled tags are those that allow you to use EL expressions. Refer to Chapter 10 for examples.

---

nest

This tag allows the nesting to be moved up by one level. All replacement tags within the current `<nested:nest>` tag will take as their base object the value indicated by the `property` attribute of this tag.

Usage Restrictions

None, but you obviously need to specify the `property` attribute in order to accomplish anything.

Attributes

There is just one attribute: `property`, which indicates the property on the current implicit base object from which all child Nested tags will use their base object.

Examples

Suppose you wanted to access a nested property using, say, `<bean:write>`, in an `<html:form>`.

```
<html:form action=...
   <bean:write property="contact.name"/>
   <bean:write property="contact.email"/>
   <bean:write property="contact.company"/>
</html:form>
```

You can use `<nested:nest>` and `<nested:write>` tags instead:

```
<html:form action=...
   <nested:nest property="contact">
      <nested:write property="name"/>
      <nested:write property="email"/>
      <nested:write property="company"/>
   </nested:nest>
</html:form>
```

In both snippets, the implicit base object is determined by the enclosing `<html:form>`. The base object is just the form bean instance associated with the form. If you wanted to change the base object, you'd have to use `<nested:root>`.

Equivalents

None.

writeNesting

This tag is used to write the current nesting level either to the response stream (i.e., the rendered HTML page), or to make it available as a page scoped variable.

Usage Restrictions

None.

Attributes

This tag has three attributes:

- `property`: The value of this attribute is blindly appended to the end of the nesting level, prefixed with a dot.
- `id`: When `id` is specified, the nesting level is not written to the final HTML page, but rather is exposed as a page-scoped `String` variable with the value of `id` serving as the name.
- `filter`: This attribute takes the value `true` or `false`. If `true`, then the nesting level is encoded. For example, the ampersand (`&`) appears as `&`. Despite what the Apache online documents say, URL encoding is not performed—for instance, spaces are not encoded as `%20%`, etc.

These attributes are best illustrated with JSP examples.

Examples

Using `writeNesting` only makes sense when it's used in conjunction with the `<nested:nest>` tag as in the following snippet:

```
<html:form ...>
   <nested:nest property="contact">
      <nested:nest property="company">
         Nesting level is: <nested:writeNesting />
      </nested:nest>
   </nested:nest>
</html:form>
```

In this example, the displayed HTML will contain the text

```
Nesting level is: contact.company
```

If we had used `<nested:writeNesting id="dummy" />`, then the output would have been blank. This is because instead of being displayed on the page, a page-scoped variable named `dummy` has been created. To display it, you need to use a scriptlet embedded within the same nested level as the `<nested:writeNesting>` that defined the variable. For example:

```
<html:form ...>
   <nested:nest property="contact">
      <nested:nest property="company">
         <nested:writeNesting id="dummy"/>
         Nesting level is: <%= dummy %>
      </nested:nest>
   </nested:nest>
</html:form>
```

Note that if you placed the scriptlet in a different nesting level, you'd get an error:

```
<html:form ...>
   <nested:nest property="contact">
      <nested:nest property="company">
         <nested:writeNesting id="dummy"/>
      </nested:nest>
      Nesting level is: <%= dummy %> //ERROR!!
   </nested:nest>
</html:form>
```

Lastly, the `property` attribute simply appends the given value to the end of the nesting level. So the code snippet

```
<html:form ...>
   <nested:nest property="contact">
      <nested:nest property="company">
         Nesting level is: <nested:writeNesting property="hello" />
      </nested:nest>
   </nested:nest>
</html:form>
```

displays on the HTML page as

```
Nesting level is: contact.company.hello
```

Equivalents

None.

root

This tag is used to specify a base object.

You use `root` either to specify a different base object from the current one (as is the case within an `<html:form>` tag) or to define a base object if none exists (as is the case anywhere outside an `<html:form>`).

Usage Restrictions

None, but you obviously need to specify the `name` attribute in order to accomplish anything.

Attributes

There is just one attribute, `name`, which is the name of the JavaBean on the current request. This bean is used as the new base object for all Nested tags enclosed by `<nested:root>`.

Examples

Suppose you wanted to access a nested property using, say, `<bean:write>`, in an `<html:form>`, but with the base object called `MyBean` instead of the base object implied by `<html:form>`. Here's how you would do it:

```
<html:form action=...
   <bean:write name="MyBean" property="contact.name"/>
   <bean:write name="MyBean" property="contact.email"/>
   <bean:write name="MyBean" property="contact.company"/>
</html:form>
```

To accomplish the same thing with the Nested tags, you'd have to use `<nested:root>`:

```
<html:form action=...
   <nested:root name="MyBean">
      <nested:nest property="contact">
         <nested:write property="name"/>
         <nested:write property="email"/>
         <nested:write property="company"/>
      </nested:nest>
   </nested:root>
</html:form>
```

As you know (see the corresponding reference), `<html:form>` defines an implicit base object, which is the form bean associated with that form. `<nested:root>` allows you to temporarily redefine the base object. This would also work outside an `<html:form>`. For example, the snippet

```
<bean:write name="MyBean" property="contact.name"/>
<bean:write name="MyBean" property="contact.email"/>
<bean:write name="MyBean" property="contact.company"/>
```

may be replaced with

```
<nested:root name="MyBean">
   <nested:nest property="contact">
      <nested:write property="name"/>
      <nested:write property="email"/>
      <nested:write property="company"/>
   </nested:nest>
</nested:root>
```

Equivalents

None.

The Tiles Tag Library

The Tiles tag library supports the Tiles plug-in, which is used to create layouts and reusable GUI components. Chapter 14 explains Tiles in detail, and you should consult it to understand how to use Tiles.

Table C-13 gives a synopsis of all tags in the Tiles tag library. Table C-13 is based on the Apache documentation. Please refer to `www.apache.org/licenses/LICENSE-2.0` for a copy of the Apache License.

Table C-13. *Synopsis of the Tiles Tag Library*

Tag	Usage/Comments
`insert`	Inserts a layout or Tile.
`definition`	Declares a layout or customized layout for later use.
`put`	Adds a Tiles attribute to the current context.
`putList` and `add`	`putList` defines a list attribute in the current context. `add` is used to add items to the list.
`get`	Writes a specified JSP or Tiles definition to the response stream.
`getAsString`	Writes a specified Tiles attribute as a literal string to the response stream.
`useAttribute`	Exposes a Tiles attribute to scriptlets or other custom tags.
`importAttribute`	Allows scriptlets or other custom tags to read Tiles attributes.
`initComponentDefinitions`	Legacy tag that allows the use of Tiles outside of Struts.

Common Attributes

Unlike the other Struts tag libraries, Tiles has no real common attribute sets. However, there are a number of common attributes. These are described in Table C-14.

Table C-14. *Commonly Used Attributes in the Tiles Tag Library*

Attribute	Description
`template/component/page`	These are synonymous, and they all refer to a JSP page containing layout information.
`beanName/beanProperty/beanScope`	These attributes are used to locate a JavaBean instance (using `beanName` and optionally `beanProperty`) on a specified JSP scope (`application`, `session`, `request`, `page`) or on the Tile's `ComponentContext` (in which case you use `beanScope=tile/component/template`—these are all synonymous). If `beanScope` isn't specified, then all scopes (first the `ComponentContext` and then all JSP scopes) are searched for the named JavaBean instance.
`role`	An authentication role (see the discussion on authentication roles in the entry for `<logic:present>`) that the remote user must have in order for the enclosing tag to be processed. If the remote user does not have the specified role, the enclosing tag is skipped.
`ignore`	If `true` (the default is `false`), then if data for the enclosing tag (which is determined using the `name` attribute for the tag; this `name` attribute will have interpretations that vary by tag) is not found, simply skip further processing for the enclosing tag. The default value of `ignore` is `false`, which means that an exception is thrown if the tag's data can't be found.
`flush`	If `true`, causes the `OutputStream` to be flushed before the tag is processed. This attribute is needed to fix problems with some servlet containers. The default is `false`.

A Note on Equivalent Tags

You should note that there are no pure JSF (or JSTL) equivalents for Tiles. So, I've omitted the "Equivalents" section on all entries.

Struts-EL Tags for the Tiles Tag Library

All Tiles tags have EL-enabled versions.

■ **Note** EL-enabled tags are those that allow you to use EL expressions. Refer to Chapter 10 for examples.

insert

This tag inserts a page or Tiles layout or controller into the enclosing page. You can customize a layout using nested `<tiles:put>` tags.

Usage Restrictions

You must specify what to insert. You do this using either the `template/component/page` synonyms or `definition` or `attribute` or `name` attributes.

Attributes

Besides the commonly used `role`, `flush`, and ignore attributes (see Table C-14), the following attributes specify what to insert:

- `template/component/page`: The name of the JSP page to insert.
- `definition`: The name of the Tiles definition to insert.
- `attribute`: The defined property referring to a JSP page or Tiles definition to insert. The use of this property assumes that you've defined the attribute elsewhere using `<tiles:put>`. Refer to Listings 14-3 and 14-4 to see how to use this attribute.
- `name`: The name of the JSP page, Tiles definition, or `<tiles:put>` defined property to insert.
- `beanName/beanProperty/beanScope`: These attributes are used to locate an object (see C-14 on how this is done). The located object is used in one of two ways: (1) If the object is an instance of `org.apache.struts.tiles.AttributeDefinition` or `org.apache.struts.tiles.ComponentDefinition`, then these represent a JSP page or Tiles definition to insert. Frankly, it is difficult to see when this option might be useful. (2) Otherwise, the object's `toString()` is called, and this is used as the name of the JSP page, Tiles definition, or `<tiles:put>` defined property to insert.

You can also call controller code to prepare data or display messages before the insert is performed. To do this, you can specify one of these attributes:

- `controllerUrl`: This can be a form handler's name or a URL.
- `controllerClass`: Any class that implements the `org.apache.struts.tiles.Controller` interface. This interface has a single `execute()` method, as described in Listing 14-9.

Examples

This code inserts a layout using `template/component/page`. Note the leading slash:

```
<tiles:insert page="/mypage.jsp" />
```

You can also similarly insert a Tiles definition:

```
<tiles:insert definition=".basic" />
```

For more examples using <tiles:insert>, refer to Chapter 14.

definition

This tag declares a layout, or a customization of it, enabling it to be reused. You call the definition from a <tiles:insert> to actually use it.

Usually, you first declare the definition in a shared JSP page, then use <%@ include file="..." %> to include that JSP in your pages that utilize that definition.

<tiles:definition> also allows you to customize an existing layout or existing definition, then reuse that customized version in one or more JSP pages of your own. You perform customization using <tiles:put> and related tags.

Notice that I haven't discussed the use of the <tiles:defiinition> tag in Chapter 14 because a much better alternative is to declare your definitions in the tiles-defs.xml file instead. This makes maintenance of your definitions easier to track, and obviates the need to use shared JSP pages.

Usage Restrictions

The id attribute is required.

Attributes

- id: Specifies a unique name for this definition.
- scope: Specifies the JSP scope of the definition, once it is created. The default scope is page. Setting the value to a scope with a longer lifetime (say session) would allow you to access that definition with <tiles:insert> throughout the user session. Of course, used unwisely, this freedom leads to "high maintenance" webapps.
- template/page: The name of the JSP page that holds the actual layout. Remember that definitions only declare a layout. The actual layout itself is contained in a JSP page. This page usually contains <tiles:insert attribute="..."/> tags to dictate placement of named items. Refer to Listing 14-3 for an example of this.
- role: A role that must match any one of the remote user's roles in order for this definition to be processed. For a brief discussion on roles, refer to the entry for <logic:present>.
- extends: The name of a "parent" definition that you want to further customize. See Listing 14-7 to see how this works.

Examples

Consider the layout page (`mylayout.jsp`) fragment here:

```
<table>
   <tr><td><tiles:insert attribute="header"/></td></tr>
   <tr><td><tiles:insert attribute="body"/></td></tr>
   <tr><td><tiles:insert attribute="footer"/></td></tr>
</table>
```

This layout specifies the relative placement of three elements: `header`, `body`, and `footer`. To use this layout, you can declare a `<tiles:definition>` like so, in a shared JSP page (`shared-layouts.jsp`):

```
<tiles:definition id=".basic" page="/mylayout.jsp" />
```

This declares the page as it is, with no customizations made. Usually, however, it makes sense to customize a layout when you declare it. In the previous example, we might want to specify the content for the elements `header` and `footer`:

```
<tiles:definition id=".basic" page="/mylayout.jsp">
   <tiles:put name="header" value="/navbar.jsp" />
   <tiles:put name="footer" value="/terms-of-use.jsp" />
</tiles:definition>
```

In this snippet, the `header` now points to `navbar.jsp` and the `footer` to `terms-of-use.jsp`. So, the remaining element, `body`, is undefined. To use the declared definition, you'd include the shared JSP and then use `<tiles:insert>`:

```
<%@ include file="shared-layouts.jsp" %>
<tiles:insert definition=".basic">
   <tiles:put  name="body"   value="hello-world.jsp" />
</tiles:insert>
```

Note that the body is now defined to point to `hello-world.jsp`.

put

`put` defines a named attribute in a layout. This allows you to customize a layout for display.

Usage Restrictions

The `put` tag must be nested within a `<tiles:insert>` or `<tiles:definition>`. You must also specify `name`, which refers to the attribute in the layout, and `content` for that attribute.

Attributes

- `name`: The name of the attribute that the `put` defines.
- `value/content`: A URL or `String` representing the content for an attribute. `value` and `content` are synonymous.
- `beanName/beanProperty/beanScope`: Used to locate the object that is interrogated in order to determine the value for this put. Refer to Table C-14 for details. These attributes can be used instead of `value` or `content`.
- `type`: Indicates the meaning of the value for this put—is it a URL to a JSP page (`type="page"` or `type="template"`) or a Tiles definition (`type="definition"` or `type="instance"`) or a `String` (`type="string"`)? If unspecified, implies `type="page"`, unless `direct="true"`, in which case, `type="string"`.
- `direct`: If set to `true`, causes `value` to be interpreted as a literal `String` instead of a reference (URL to a JSP page or Tiles definition) to be evaluated. The default is `false`.
- `role`: A role that must match any one of the remote user's roles in order for this put to be processed. For a brief discussion on roles, refer to the entry for `<logic:present>`.

Examples

Refer to the examples in the entry for `<tiles:definition>` to see how `put` is typically used. You may also specify the value of a put in its body like so:

```
<tiles:put name="myPageTitle">Here's looking at you!</tiles:put>
```

This form is useful when you need to specify text that doesn't conform to XML's attribute format.

putList and add

`putList` creates a list of items, and specifies a name for that list. One or more `add` tags are nested within a `putList`, and are used to add elements to the list.

Usage Restrictions

You must specify the `name` attribute for the `putList`.

Attributes

`putList` has just one attribute, `name`, which is the name of the list. The `add` tag has exactly the same attributes as `<tiles:put>`, except that for the `name` attribute.

Examples

You should understand that `putList` and `add` can't be used to easily customize layout, as you could with `<tiles:put>`. For example, you'd expect to be able to define a customization of a layout like so:

```
<tiles:insert page="mylayout.jsp">
   <tiles:putList name="sideBarItems">
      <tiles:add value="/sidebar1.jsp"/>
      <tiles:add value="/sidebar2.jsp"/>
      <tiles:add value="/sidebar3.jsp"/>
   </tiles:putList>
</tiles:insert>
```

Then, in the layout page (`mylayout.jsp`), you'd expect to do this:

```
<tiles:importAttribute />
<logic:iterate name="sideBarItems" id="item">
   <tiles:insert  name="item"/> //ERROR!!
</logic:iterate>
```

This won't work: you can't nest `<tiles:insert>` inside a `<logic:iterate>` (a `ServletException` is thrown). You'd have to resort to using scriptlets that loop through the items:

```
<tiles:importAttribute />
<%
  for(Iterator items =  sideBarItems.iterator(); items.hasNext();){
    String item = items.next().toString();
%>
    <tiles:insert  name="<%=item %>"/>
<% } %>
```

which is much less readable.

get

`get` reads the `name` attribute from the `ComponentContext` for the Tile (placed there using `<tiles:put>` or `<tiles:putList>`), interprets the value as a JSP page or a Tiles definition, and writes the output of either to the response stream.

Usage Restrictions

You need to specify what to display, so the `name` attribute is required.

Attributes

- `name`: The name of the Tiles attribute to include.
- `ignore`: If the attribute specified by `name` can't be found, `ignore="true"` causes a silent failure. Otherwise, an exception is thrown, which is the default behavior.
- `flush`: This has its usual meaning (see Table C-14).
- `role`: This has its usual meaning (see Table C-14).

Examples

Consider the snippet

```
<tiles:insert page="one.jsp">
   <tiles:put name="myTilesAttr" value="two.jsp"/>
</tiles:insert>

<tiles:get name="myTilesAttr" />
```

This will cause the contents of `two.jsp` to be displayed after the contents of `one.jsp`.

getAsString

This tag is similar to `<tiles:get>`, but the Tiles attribute is interpreted as a literal string. This is useful to display static text (used for example, in `<title>` sections of web pages).

Usage Restrictions

You need to specify what to display, so the `name` attribute is required.

Attributes

- `name`: The name of the Tiles attribute to display. If this isn't found, an exception is thrown.
- `flush`: This has its usual meaning (see Table C-14).
- `role`: This has its usual meaning (see Table C-14).

Examples

Consider the snippet:

```
<tiles:insert page="one.jsp">
   <tiles:put name="myTilesAttr" value="two.jsp"/>
</tiles:insert>

<tiles:getAsString name="myTilesAttr" />
```

This will cause the string "two.jsp" to be displayed after the contents of `one.jsp`.

useAttribute

This tag exposes a Tiles attribute as a Java variable with a given scope. Contrast this with `<tiles:importAttribute>`.

Usage Restrictions

You need to specify the `name` of the Tiles attribute to expose.

Attributes

- `name`: The name of the Tiles attribute to expose. Required.
- `id`: The name by which the Tiles attribute will be known to scriptlets or other tags. If omitted, `id` defaults to the name of the Tiles attribute.
- `scope`: The JSP scope of the exposed variable (`page`, `request`, `session`, or `application`). The default is `page` scope.
- `classname`: The classname of the exposed variable.
- `ignore`: If `true`, fails silently if the Tiles attribute can't be found. The default is to throw an `exception`.

Examples

Consider this snippet:

```
<tiles:insert page="mylayout.jsp">
   <tiles:put name="myTilesAttr" value="one"/>
</tiles:insert>
<tiles:useAttribute name="myTilesAttr" id="x"/>
<%= x %>
```

Here, we've exposed the Tiles attribute `myTilesAttr` as a scripting variable named `x`.

importAttribute

`importAttribute` copies specified Tiles attributes (declared with `<tiles:put>` or `<tiles:putList>`) into the specified scope. Recall that each Tile stores its attributes into a `ComponentContext` instance, and not directly on the request, session, page, or application scope. Because of this, you need to copy attributes from the `ComponentContext` into the request, session, page, or application scope in order for non-Tiles tags to read the attributes. `importAttribute` does just this.

Usage Restrictions

`importAttribute` needs a `ComponentContext` to read from, and if an instance isn't found, an exception is thrown. This means that you must place `<tiles:importAttribute>` after a `<tiles:insert>` or within a page called by a `<tiles:insert>`, to ensure that the `ComponentContext` has been created.

Attributes

- `name`: Specifies the name of the attribute to copy over. If omitted, all attributes of the `ComponentContext` are copied to the prescribed scope.
- `scope`: Specifies a scope (request, session, page, or application) to copy attributes to. If omitted, the page scope is used.
- `ignore`: If the named attribute can't be found, an exception is thrown by default. Setting `ignore="true"` causes `importAttribute` to fail silently.

Examples

Consider this snippet:

```
<tiles:insert page="mylayout.jsp">
   <tiles:put name="object1" value="one"/>
   <tiles:put name="object2" value="two"/>
   <tiles:put name="object3" value="three"/>
   <tiles:put name="object4" value="four"/>
</tiles:insert>

<bean:write name="object1"/> //ERROR!!
```

The error occurs because `<bean:write>` will not be able to find `object1` in any scope. For this to work, you need to use `<tiles:importAttribute>`:

```
<tiles:importAttribute />
<bean:write name="object1"/> //OK
```

initComponentDefinitions

This tag is used to initialize the Tiles subsystem. Recall that Tiles was originally not part of Struts, and can actually be used by itself. So, this tag is only useful if you're using Tiles outside of Struts. For example, if you write your own servlets, this tag is useful because it allows you to initialize the Tiles subsystem without having to rely on Struts to do it for you. Frankly, Tiles is now so much integrated with Struts that this tag is no longer very practical.

APPENDIX D

Answers

Listed in this appendix are answers to questions raised in the main text.

Chapter 1: Introduction

Question: If pieces of code that do different things are bundled separately and given standardized ways of communicating between themselves, people can maintain or work on some parts of the webapp without having to worry too much about other parts. Can you see why?

Answer: An analogy might help: think of the last time you filled in your income tax form. Not a pleasant task, but imagine how much worse it would be if you had to declare your income *without* such a form for guidance. Very likely, you'd miss something and there would be questions and answers bouncing back between you and your favorite IRS agent. Not a very efficient way to collect or pay taxes! The obvious magic behind tax forms is that they standardize communication between you and the IRS.

Another big advantage of using a form is that you don't have to know how the IRS operates. For example, you don't have to know how your income tax is processed. You only have to know how to fill the tax form. So, with tax forms, you can efficiently communicate with the IRS for the purpose of paying income tax, without having to know too much about them, or guess what information they want from you.

The same idea works for code, and applies to people who build different parts of an application. Partitioning code along lines of functionality and giving the resulting code units standardized ways of communicating between themselves (in Java terms, *interfaces*), simplifies the code required (just as you don't need to know how the IRS works) and facilitates communication between the application builders, making it easier for them to build or maintain the application in parallel.

In summary, the separation of the code by functionality *and* the use of standardized methods of communication imply easier development and maintenance because

- A code module needs to know less about the internals of a module it communicates with.
- The need to know less translates to simpler code.

Chapter 3: Understanding Scopes

The answers that follow address these questions for application, session, request, and page scopes:

- Audrey is the first person to view `First.jsp`. What will she see?
- Brenda next views `First.jsp` from a different machine. What does she see?
- If Audrey again views `First.jsp` *after* Brenda, what will she see?
- What if Brenda now loads `Second.jsp` directly?

Answers for application scope:

- Audrey sees `12`. Each page appends its number to `myVariable`.
- Brenda sees `1212`. Brenda appends a `12` to Audrey's `12`.
- Audrey sees `121212`.
- Brenda sees `1212122`. The last `2` does not have an accompanying `1` because Brenda does not load `First.jsp`.

Answers for session scope:

- Audrey sees `12`. Each page appends its number to `myVariable`.
- Brenda sees `12`. Brenda gets a clean copy of `myVariable`.
- Audrey sees `1212`, since `myVariable` exists for the whole session.
- Brenda sees `12122`. The last `2` does not have an accompanying `1` because Brenda does not load `First.jsp`.

Answers for request scope:

- Audrey sees `12`. Each page appends its number to `myVariable`. Request-scoped variables survive a page forward.
- Brenda sees `12`. Brenda gets a clean copy of `myVariable`.
- Audrey sees `12`, since she's seeing a new copy of `myVariable`.
- Brenda sees `2`. The `2` does not have a `1` preceding it because Brenda does not load `First.jsp`.

Answers for page scope: Audrey and Brenda both just see `2` in each scenario, since page-scoped variables exist only on the last page called.

The variable `x` in `<% int x = 7; %>` has an implicit page scope.

Chapter 5: The MVC Design Pattern

Requirement 1: A page to collect the information about a contact

View = code that displays a form for data entry

Requirement 2: Code to check contact details (postcode, email, or required fields) after submission and display errors

Controller = code that checks the data, and display errors if any

Requirement 3: Code to store and retrieve data into a database

Controller = code that calls the Model to store data

Model = code that actually stores the data

Chapter 6: Simple Validation

Simple validation quiz answers:

Q1: Suppose validating a certain field requires a numerical calculation. Would this qualify as a simple validation? You should give clear reasons for your answer.

Answer: Generally, this would not qualify as a simple validation. One good rule for identifying a simple validation is if the validation in question could be made generic (like a check for email format). In this case, arbitrary numerical calculations are considered "data transformations" and should therefore qualify as a complex validation.

Q2: When is `validate()` called? Before Struts populates the `ActionForm` or after?

Answer: `validate()` is called *after* Struts populates your `ActionForm` subclass.

Q3: If `validate()` returns an empty `ActionErrors` instance, will the form be redisplayed?

Answer: No, the form will not be redisplayed. There were no errors!

Q4: Rewrite Listing 6-1 so that Struts only displays the error message for the *first* validation error encountered.

Answer: You could insert a `return errors;` after each validation.

Q5: If you changed the name of one of a form's properties (say `zipcode` to `postcode`), would you have to change the corresponding error key for `ActionErrors`, that is, `errors.add("zipcode",...)` to `errors.add("postcode",...)`?

Answer: You are not *required* to make the change, because the error keys are independent of the form property names. However, it would be *wise* to do so, to avoid confusion.

Q6: If your answer to Q5 was "yes," what other changes would you have to make?

Answer: If you did make the change, you'd also have to change the `property` attribute on the `<errors>` tag. In other words, you'd have to change `<html:errors property="zipcode"/>` to `<html:errors property="postcode"/>`.

Q7: If there were more than one validation error for a single property, which error message (if any) do you think would be displayed?

Answer: If a property had more than one error message, all error messages for that property would be displayed. This is often confusing, so the trick in Q4 is useful.

Lab 8: Contact Entry Page for LILLDEP

Step 1: Which libraries would you use?

Answer: The HTML and Bean tag libraries

Step 5: What is the name of the handler for this form?

Answer: `ContactFormHandler`

Lab 9a: Configuring LILLDEP

Step 2: What should the name of the form handler be?

Answer: There are two ways to answer this question:

- The name by which this form will be known to Struts tags is `ContactFormHandler`, so the `path` attribute is declared as `path="/ContactFormHandler"`. (Throughout this book, this is what I mean when I refer to the "name" of a form handler.)

- The `name` attribute of the form handler is `ContactFormBean`. This is simply a reference to the form bean for this handler. It is *not* the name by which this handler is called in the Struts tags.

Step 5: Give a value for the form handler's `input` attribute. Where is this used? What happens if you omit this attribute from the form handler's declaration?

Answer: The form handler's `input` attribute should be `/full.jsp`. This value is used explicitly whenever `mapping.getInputForward()` is called, for example, in `ContactAction.java`. It is also used implicitly by Struts in order to redisplay the input page if there's a simple validation error. If this value is omitted, then the input page does not display. A generic servlet error page is displayed instead.

Step 6: Create a forward for this form handler to the page `full.jsp`. What should be the `name` attribute of this forward? (Hint: Check the code for the `Action` subclass.) What happens if you omit this forward declaration?

Answer: The `name` attribute of the forward is `success`. If this value is omitted, then the "next" page does not display. A generic servlet error page is displayed instead.

Lab 9b: The MNC Page

Step 2: Add a new form handler to `struts-config.xml` to accept data from `MNC.jsp`. The (single) forward should point back to `MNC.jsp`. What should the name of the forward be?

Answer: The name of the forward is `success`.

Chapter 10: More Tags

Flashback Quiz: Can you remember what the counterparts of `var` and `items` were on `<logic:iterate>`?

Answer: `var` exposes a single element of the iteratable object, so its counterpart on `<logic:iterate>` is `id`. And, since `items` refers to the name of the iteratable object, its `<logic:iterate>` counterpart is `name`.

Now You Do It: Construct JSTL equivalents for `equal`, `present`, `empty`, `lessThan`, `lessEqual`, `greaterThan`, and `greaterEqual`, and the negatives for all of the above (`notEqual`, `notPresent`, etc.).

Answer: All the above should be trivial to construct. For example, the JSTL test for

```
<logic:lessThan name="myConstants" property="pi" value="3.14">
```

would be

```
test="${myConstants.pi < 3.14}"
```

The other tags are similarly constructed. The exception is `<logic:present>`. This tag tests for the presence of an item in the current request. There are two *nonequivalent* ways to test for this. You could use

```
test="${myVariable != null}"
```

to test if the variable `myVariable` reference exists. Or you could use

```
test="${not empty myVariable}"
```

This latter check will return `true` not only if the `myVariable` does not exist but also if the `myVariable` exists but is `empty`, meaning it is a zero length `String`, or an empty array or `Collection`.

Another Struts tag with a not-so-obvious JSTL replacement is `<logic:messagesPresent>`. This tag tests for the presence of a Struts message. For example:

```
<logic:messagesPresent property="emailMsg">...
```

checks for the presence of a message named `emailMsg`. In order to emulate this tag using JSTL, we have to know where Struts messages are stored on the request object. A quick comparison of the Struts source code files involved (`org.apache.struts.taglib.logic.PresentTag`, `org.apache.struts.taglib.logic.MessagesPresentTag`, and `org.apache.struts.taglib.TagUtils`) shows that the tests are essentially the same. So, the previous two solutions would also apply to `<logic:messagesPresent>`.

Lab 10a: The LILLDEP Full Listing Page

Step 4: Will you need a form bean for this action mapping?

Answer: No, because we're not processing user input. Instead, we're using this handler to generate the listing.

Lab 10b: Simplifying ContactForm

Step 1: If you compiled and deployed LILLDEP now, would it work? Why or why not?

Answer: No, it would not work. You also have to amend the JSPs to use the relevant `<nested>` tags, *or* change the `property` attributes to use nested properties.

Step 2: In either approach, what is the new base object?

Answer: In either approach, the new base object is `contact`. For example, using nested properties, the new property declaration for the email address is

```
property="contact.email"
```

Lab 11: Importing Data into LILLDEP

Step 1: Do you need to implement the `validate()` or `reset()` method?

Answer: Yes, you could use `validate()` to ensure that the uploaded file has a nonzero size, and has the right extension (`.csv` or `.CSV`). You could use `reset()` to call `destroy()` on the `FormFile` instance.

Step 2: What should the `property` attribute of `<html:file>` be?

Answer: `<html:file property="file" />`

Step 4: Define a new form handler to handle the importing. What is the `path` attribute of the form handler?

Answer: `path="/ImportForm"`

Chapter 13: Review Lab

Question: If you turn the company name into a link, how will you determine which company was clicked? (*Hint*: Look at the source code for `BaseContact`.)

Answer: You'd embed the ID of the contact (determined using `Contact.getId()`) as a parameter in the link. This link could point to a form handler that

- Creates the appropriate `Contact` (you must know how to use the `Criteria` class to do this—see Appendix A) based on the ID embedded in the link (refer to the functions on `HttpServletRequest` to learn how to do this—see Appendix B)
- Puts this `Contact` instance into the `ContactForm`
- Forwards to `full.jsp` for display

Question: You obviously want to reuse `full.jsp` to display the data that's going to be edited. Do you need to make any changes to it to support updating contact information? Why?

Answer: No changes are needed to `full.jsp`, since the `Contact` instance's `save()` function (actually, on `BaseContact`), correctly handles updating.

Question: Can you similarly reuse `ContactForm` and `ContactAction`? Do you need to make changes to them to support updating?

Answer: Yes, we can reuse both. No changes needed, since the `Contact.save()` correctly handles updating. This is the power of using Model classes instead of embedding SQL in your `Action` subclasses.

Question: What other classes would you need to complete the editing facility? (*Hint*: What creates the populated form for editing?)

Answer: We need a new `Action` to create and populate the form. See the answer to the first question for details.

Chapter 14: Tiles

Quick Quiz: What advantage does this approach (Listings 14-5 and 14-6) have over the earlier one (Listing 14-4)?

Answer: Listing 14-4 applies a layout to a single page. It can't be reused. More precisely, you can reuse the *page* in your webapp of course, but you can't reuse the *layout* in your other JSP pages. To reuse the layout, you'd have to define the layout in your Tiles definitions file. This is what Listing 14-5 does; it defines a layout that may be applied to any number of pages, as it is in Listing 14-6.

Chapter 15: The Validator Framework

Question: In most cases, putting in a `validate()` function should suffice. Use the second alternative only after much thought, since validators should be *generic*, if only to the problem domain of your web application. Otherwise, you'd run into potential maintenance issues. Can you see why?

Answer: If a particular validation is going to be used only once, it makes no sense to write an extension to the Validator framework. This is because the maintenance overhead for such an extension (you need to declare the new validator in `validation-rules.xml`, create the necessary Java code, and use the validator in `validations.xml`) is much higher compared to using a `validate()` function.

Of course, if the same validation is used many times over (i.e., the validator is *generic*), then the cost of repeating the code in `validate()`, or even refactoring necessary validator code in a separate class, is higher. The latter option might seem to be lower in cost, but you'd still have to implement `validate()` on each form you use.

Chapter 17: Potpourri

Question: As you should know by now, `unspecified()` handles the case where the requested function does not exist on `PrintingAction`. (Question: how can this happen?)

Answer: It can happen if there's a bug in the JSP code (that is, the JSP code omits the `dispatch` parameter in the URL), or if the user manually keys in an incorrect URL (again, omitting the `dispatch` parameter in the URL).

`unspecified()` *does not* handle a `dispatch` parameter value for which the corresponding function does not exist. In this case, an `Exception` is thrown and unless you've caught it using a global (or local—see Chapter 18) exception handler, you'll see the standard JSP error page. An alternative to using exception handlers is to extend `LookupDispatchAction` (as I did in Lab 14's `TilesLookupDispatchAction`) to include an `unknown()` function that handles this possibility.

Chapter 20: JavaServer Faces and Struts Shale

Quick Quiz: How would you do the same thing in Struts?

Answer: Use `LookupDispatchAction`. The JSF approach is much easier to use. Instead of going through the trouble of subclassing `LookupDispatchAction` and declaring special parameters in `struts-config.xml`, all you need to do with JSF is to tell JSF which function to call in the `commandButton`'s `action` attribute.

Index

Symbols

#{} versus ${}, 342

${...} delimiters, evaluating EL expressions with, 120

${...} versus #{...}, 340

‘ (single quotes), escaping in JSF, 325

* (asterisk) wildcard, using, 262

. separator, relationship to Tiles layouts, 167

. tiles-def.xml, with new definition, 167

[] (square brackets), declaring indexed properties with, 223

{0} wildcard, using, 262

| (pipe symbol), using as separator, 140

<%@ page, using with JSPs for character encoding, 147

Numerics

100 Celsius
- converting to 212 Fahrenheit, 31
- converting to 373 Kelvin, 31

212 Fahrenheit, converting 100 Celsius to, 31

373 Kelvin, converting 100 Celsius to, 31

A

a tiles-documentation.war filecontents of, 162

absolute() function of Scroller, explanation of, 275

AbstractViewController class, implementing ViewController with, 346

Accessibility attribute set, attributes and usage of, 384

Action class, description of, 377

Action Controller class
- role in business logic, 67, 70
- statelessness of, 68
- subclassing, 69–70

Action subclasses, overriding execute() method in, 69

<action> tags
- attributes of, 98
- declaring for MappingDispatchAction subclass, 256
- using <forward> tags with, 100

ActionErrors
- relationship to validate(), 55
- using, 57, 60
- using with Struts-Faces integration library, 342

ActionForms
- anatomy of, 54, 57
- casting to RegistrationForm, 72
- description of, 378
- subclass of, 54

ActionForm properties, type conversion idiom for reading of, 241

ActionForm subclasses
- coding for checkbox EL tag, 392
- decoupling Action from, 242
- implementing for Registration webapp, 55
- mapping to names in struts-config.xml, 93
- naming with form beans, 95
- reading properties in, 241
- relationship to business logic, 67
- role in uploading files, 130
- simplifying, 126

ActionForms
- versus backing beans, 321
- conceptualizing in JSF, 311
- versus DynaActionForms, 226
- versus UI trees, 314

ActionForward
instantiating in business logic, 75
versus Java String, 322
ActionMapping class, overview of, 377
<action-mappings> enclosing tag, defining form handlers in, 98
ActionMessages
adding to ActionErrors instances, 57
holding error messages in, 73
overview of, 376
storing, 377
string argument of, 59
ActionServlet, relationship to Struts, 48
add EL tags, using, 462–463
add() function for ActionMessages, 57
addObjectCreate(path, bean) function, using with Digester, 289
addParameter() functions, calling with Digester, 304
addSetProperties(path) function, using with Digester, 289
Ajax, relationship to Shale, 354
alphanumeric strings, Java handler for validation of, 213–214
alphanumeric validator, declaring, 215
Ant website, 358
Apache Ant website, 35
Apache Commons logging, significance of, 260. *See also* logging
Apache Commons Validator classes, functions associated with, 214
Apache Commons' BeanUtils project, website for, 237
Apache Struts website, 6
Apache's Digester. *See* Digester
Application scope, explanation of, 17
Application.properties, role in localizing output, 150–151
Application.properties files
processing, 151
specifying in JSF, 324
applications, factors involved in internationalization of, 143
Apply Request Values phase in JSF, explanation of, 316
arithmetic operators, using with EL expressions, 120
array lengths, setting at runtime, 224
array size, specifying for indexed properties, 223
ASCII versus Unicode, 144
asterisk (*) wildcard, using, 262
attribute sets
Accessibility attribute set, 384
Error Style attribute set, 386
Event Handler, 383
Initial Bean attribute set, 385
for Logic tag library, overview of, 436
Rendering attribute set, 385
Struts attribute set, 385
attributes, 24
for add tags, 462
for definition tag, 460
for empty tag, 438
for equal tag, 439
for forward tag, 440
for get tag, 464
for getAsString tag, 464
for greaterEqual tag, 442
for greaterThan tag, 442
for importAttribute tag, 466
for insert tag, 459
for iterate tag, 443
for lessEqual tag, 442
for lessThan tag, 442
for match tag, 445
for messagesNotPresent tag, 446
for messagesPresent tag, 446
for nest tags, 453
for notEmpty tag, 438
for notEqual tag, 439

for notMatch tag, 445
for notPresent tag, 447
for option and options tag, 412
for option, options, and optionsCollection tags, 412–413
for present tag, 447
for put tag, 462
for putList tag, 462
for redirect tag, 449, 450
for root tags, 456
for select tag, 412
for Tiles tag library, 458
for useAttribute tag, 465
for writeNesting tag, 454
for base tag, 387
for button tag, 388
for cancel tag for HTML tag library, 389
for checkbox tag, 390
for cookie tag, 422
for define tag, 424
for errors tag, 393
for form tag, 395
for header tag, 422
for hidden tag, 397
for html tag, 398
for image tag, 399
for img tag, 400, 402
for include tag, 426
for javascript tag, 402, 404
for link tag, 404
for message tag, 427
for messages tag, 405–406
for multibox tag, 407
for page tag, 430
for parameter tag, 422
for password tag, 419
for radio tag, 409
for reset tag, 409
for resource tag, 431
for rewrite tag, 410
for size tag, 432
for struts tag, 433
for submit tag, 418
for tags used in conditional processing, 113
for text tag, 419
for textarea tag, 420
for write tag, 434, 435

B

backing beans. *See also* user backing bean
versus ActionForms and Actions, 321
altering appearance of pages with, 331
as POJOs (Plain Old Java Objects), 344
base classes, subclassing with Java handler classes, 26
Base Object attribute set, relationship to Logic tag library, 436
base objects, specifying, 456–457
base tag in HTML library, usage of, 89
<base> tags, creating, 386–387
Bean class, using with DefaultDynaFormsLoader, 302
Bean tag library
defining prefix for, 83
description of, 79
overview of, 90, 421
Bean.java code for DefaultDynaFormsLoader, 298, 300
<bean:> tags, replacing with <c:> tags, 121
<bean:message> tags, displaying static text with, 84
<bean:write> tags
Java handler class used with, 27
using with iterators, 111
using property attribute with, 111
beans, saving in Hibernate, 369
beans object, using with DefaultDynaFormsLoader, 302
Beans.java code for DefaultDynaFormsLoader, 297–298

Find it faster at http://superindex.apress.com

BeanVaildatorForms, hidden power of, 235
BeanValidatorForm JavaDoc, Web resource for, 237
BodyContent, getting instance of, 28
BodyTagSupport example, 26, 28
browsers, selecting locales from, 151
business logic
  processing, 67
  in Registration webapp, 70, 73
  tasks involved in, 67
  using Action Controller class with, 67, 70
button tag in HTML library, using, 89, 388–389
buttons, displaying if clicked, 409

C

<c:> tags, using, 121
Cancel button, displaying, 389
cancel tag in HTML library, using, 89, 390
cancelled() function, using with LookupDispatchAction class, 253–254
Cascading Style Sheets (CSS), 168
<c:forEach> tags, using, 122–123
ChangeLocale form handler, declaring in struts-config.xml, 153
character, definition of, 143
character encodings
  definition of, 144
  processing input for, 146–147
  using consistently, 147
character set, definition of, 144
check box input fields, rendering, 407–408
check boxes, adding to listing.jsp, 236
checkbox tag in HTML library, using, 89, 390
<choose>...<when> tags, using, 123
<c:if> tag, using, 123
classes. *See also* Model classes
  javax.servlet.http.HttpServletRequest class, 375
  javax.servlet.http.HttpSession class, 376
  org.apache.struts.action.Action, 377, 378
  org.apache.struts.action.ActionForm, 378
  org.apache.struts.action.ActionMapping, 377
  org.apache.struts.action.ActionMessage, 376
  org.apache.struts.action.ActionMessages, 377
  org.apache.struts.action.ExceptionHandler, 380
  org.apache.struts.tiles.ComponentContext, 380
  org.apache.struts.upload.FormFile, 379
  representing events in, 317
className information, storing with Digester, 304
Clay, reusable views with, 354
clicked buttons, displaying, 409–410
client-side validations, creating, 205, 351
code reuse, promoting with form-handler declarations in struts-config.xml, 103, 104
Collection facility, parts of, 267
collection ID, storing on session object, 273
Collection instances
  saving and reusing for performance optimizations, 414
  storing in session scope, 273
Collection Listing page, creating, 271
Collection page, creating, 269, 271
collection size, exposing, 432
collection's listing page, editing and navigating, 275–276
collections
  adding contacts to, 273–274
  removing selected contacts from, 272–273
command parameter
  checking for presence of, 185
  using with DispatchAction class, 254
Commons Logging, significance of, 260

Commons Validator framework project website, 218
commons-validator.jar file, contents of, 196
comparisions, using with EL expressions, 120
Comparison tags, effects of, 114
compile.bat, testing temperature conversion tag with, 32
complex validation
  in business logic for Registration webapp, 70
  explanation of, 53
  performing in business logic, 73–74
  performing in JSF, 329
  role in business logic, 67
complex validation failure, signaling with saveErrors(), 74
ComponentContext instance, using with Tiles controllers, 170
components. *See* Tiles components
conditional evaluation, performing with EL expressions, 120
conditional processing, performing with Logic tag library, 113–114
config objects, configuring for use with DynaForms plug-in, 305
constants, using in Validation framework, 204–205
contact entry page, creating for LILLDEP, 90
ContactActions
  changing for Find facility, 190–191
  implementing for LILLDEP, 76
ContactForms
  versus BeanValidatorForm, 235
  creating for LILLDEP, 61, 64
  migrating to use Validator framework, 218
  simplifying, 126
ContactPeer class, find() function for, 189
contacts
  adding to collections, 273–274
  deleting from listing.jsp, 237
  editing in LILLDEP's database, 157
  loading into LILLDEP, 140, 142
  removing from collections, 272–273
  selecting for deletion in LILLDEP, 236
Contacts iterator, using, 110
contentType default setting, description of, 100
controller section, declaring, 100
<controller> element, including for Struts-Faces integration library, 339
<controller> tag of struts-config.xml, restricting size of uploaded files with, 133
cookie tags
  in Bean tag library, usage of, 90
  in EL, 422–423
Core tag library in JSF, description of, 320
country property, using with <formset> tag, 217
Craig McClanahan's blog, 6
createFormBeanConfig() function, using with DefaultDynaFormsLoader, 302
creditCard validator, variables and trigger for, 206
CSS Zen Garden website, 358
CSVIterator class, using, 141
custom tags
  definition of, 14, 23
  processing, 24–25
  subclassing without bodies, 26

D

data input tags, using with Registration webapp, 86–87
data object classes, creating with Lisptorq, 361
data persistence, approaches to, 50
data source references, declaring in web.xml, 353
data transformation
  in business logic for Registration webapp, 70
  performing in business logic, 67, 74
database, initializing for Lisptorq, 365

Find it faster at http://superindex.apress.com

database schema, writing for Lisptorq, 363
database settings, specifying for Lisptorq, 364
date validator, variables and trigger for, 206
declarations, advisory about sharing of, 265
DefaultDynaFormsLoader, implementing, 294, 302
DefaultDynaFormsLoader.java, 294, 297
define tags
  in Bean tag library, 90
  in EL, 423, 425, 460–461
definitions-config parameter, passing to org.apache.struts.tiles.TilesPlugin, 162
DeleteContactsAction, implementing, 237
depends attribute, adding to <validator> declaration, 216
Derby SQL
  as embedded database, 364
  relationship to Lisptorq, 362
derby-home, using with LILLDEP, 372
design patterns
  definition of, 37
  resources for, 37
destroy() function
  role in uploading files, 130
  using with PlugIn interface, 285
detailed() function, implementing in DispatchAction class, 255
Dialog Manager in Shale, overview of, 348, 350
dialog-config.xml file, transition information in, 350
Digester
  calling addParameter() functions with, 304
  calling setXXX functions for DynaForms plug-in with, 304
  creating single user with, 288
  handling multiple user declarations with, 289, 291
  reading XML with, 288, 291
  storing className information with, 304
  Web resource for, 305
DispatchActions
  using, 254–255
  using with collections, 275–276
DispatchAction subclass, using, 255, 258
Display.jsp code sample of iterator, 110
.do prefix, using with handlers, 85
doAfterBody() function, relationship to BodyTagSupport, 27
dotted notation, using with property keys, 59
double,float,long,integer,short,byte validator, variables and trigger for, 206
DownloadAction class
  subclassing, 244–245
  using, 243, 245
DTD (Document Tag Definition)
  for Tiles, 163
  for Validator framework, 197–198
Duplicate ID error, throwing in JSF migration, 342
DynaActionForms
  accessing list of supported types for, 237
  versus ActionForms, 226
  Java classes and primitive types supported by, 223
DynaActionForm Subclasses code sample, 283
DynaForms plug-in
  description of, 277
  features of, 280
  implementing, 280, 285, 287
  passing ModuleConfig instance to, 283
  test driving, 302
  using <set-property> tags with, 303, 305
DynaFormsLoaderFactory, implementing, 291, 294
DynaFormsPlugIn, functions contained in, 287
dynamic form beans, creating for entities, 278–279. *See also* form beans
dynamic forms
  creating, 221
  declaring, 221, 225

declaring with LazyValidatorForm, 233
declaring with LazyValidatorForms, 234
definition of, 221
disadvantages of, 226
guidelines for use of, 227
reading and writing data from, 225
using with Registration webapp, 228, 232
validating, 227
dynamic properties, accessing, 225–226
dynamic Views, creating in JSF, 331

E

Eclipse website, 35, 305
edit facility, implementing for LILLDEP, 158
EJBs (Enterprise JavaBeans), relationship to web applications, 4
EL (expression language)
overview of, 119–120
relationship to JSTL, 107
using dynamic forms with, 226
EL expressions, evaluating, 120
EL extensions, overview of, 124
EL syntax versus JSF expression language syntax, 321
EL tags
migrating, 340
using, 432, 434
EL tags for HTML tag library, html tag, 398–399
EL-enabled tags, explanation of, 452
email validator, variables and trigger for, 206
empty conditional processing tag, meaning of, 114
empty EL tags, using, 437, 439
empty operator, using with EL expressions, 120
empty tag, effect of, 114
entities, hierarchy of, 278
environment entries, relationship to JNDI, 353
equal comparision tag, effect of, 114
equal EL tags, using, 439–440
error messages
displaying, 406
displaying in JSF, 330
displaying with ActionErrors, 57, 60
displaying with errors EL tags, 392, 394
holding in ActionMessages, 73
iterator for, 405, 407
in user_exists message resource file, 329
Error Style attribute set, overview of, 386
errors
checking in Login Tiles component, 186
displaying in Registration webapp, 87–88
errors tag in HTML library, usage of, 89
Event Handler attribute set, attributes and triggers for, 383
event listeners, registering for events, 320
events
flagging, 319
representing by Java classes, 317
events and event listeners in JSF
explanation of, 315
overview of, 316
exception handler
creating for Collection page, 271
defining in struts-config.xml, 93
defining, 96
ExceptionHandler, description of, 380
exceptions
printing with trace() function of Log instance, 260
throwing with functions of PropertyUtils class, 240
execute() functions
using to create Tiles controllers, 170–170
using with Tiles components, 187
execute() method, overriding in Action subclass, 69, 70

Find it faster at http://superindex.apress.com

exists() function
implementing in Lisptorq, 362
using with Torque, 366
Extended Property attributes, relationship to Logic tag library, 436–437
extends attribute, using with entities, 279–280
extensions to Struts, declaring in struts-config.xml, 94
external pages, using include EL tags with, 426

F

<f:loadMessages> versus <s:loadMessages>, 339
<f:view> tag, including in JSF migration, 340
*.faces, making entry points forward to, 341
faces requests, relationship to JSF pages, 315
faces-config.xml
configuring for JSF, 323–324
declaring managed beans in, 321
FacesContext object, using in JSF, 329
fields, validating conditionally, 207
Figures
ActionErrors tie error messages to View component, 60
Add Contacts page, 274
adding new locale with Mozilla, 152
Apache Tomcat welcome page, 9
Application.properties files, 150
Collect page showing collections, 269
Collection facility navigation button, 268
Collection Listing page, 272
compose page for uploading files, 134
Contact Editing page, 276
custom tag processing flow, 26
data flow from form to Action, 68
errors displayed, 88
Find facility in JSP pages, 188
form for Monkey's name and Banana species, 116
form to user redisplayed with error message, 71
Hello.jsp viewed in web browser, 14
hierarchy of entities, 278
HttpServletRequest and HttpSession, 12
input field for file uploading, 129
"You're Registered" page, 70
JSF's request processing lifecycle, 314
LILLDEP main page, 45
LILLDEP start page, 62, 103
LILLEP in Malay, 154
Login component control flow, 173
login form for Login Tiles component, 174
logoff view for Login Tiles component, 175
MNC data entry page, 105
ModuleConfig, FormBeanConfig, and FormPropertyConfig, 282
MVC design pattern, 38
MVC maps into Struts, 48
MyActionForm, Monkey, and Banana class diagram, 115
new Collection page, 270
output page for uploading files, 134
page processing lifecycle, 81
Registration webapp main page, 228
registration.jsp, 83
relative placement of header, body, and footer in simple-layout.jsp, 164
Scopes web application start page, 19
subclassing ActionForm, 54
Tomcat starting up, 9
treeview in MyFaces, 313
user form for Login Tiles component, 175
WAR-file structure, 11
web application frameworks, 4
file tag in HTML library, usage of, 89
file uploads
HTML element for, 129
of many files at once, 131, 133
restricting size of, 133
uploading any number of files, 133, 140

FileStreamInfo class, using with DownloadAction class, 243
FileUploadForm
code for, 131–132
uploading files with, 135
Find facility, deploying and testing, 192
Find function, adding to full.jsp and mnc.jsp, 187, 192
Find tile, placing in JSPs for Find facility, 192
First.jsp listing, 18
flow control tags, overview of, 114
focus= “userid” attribute, using with <html:form> tag, 86
<form-bean> tag, attributes of, 96
form beans. *See also* dynamic form beans
declaring, 95–96, 221–222
defining for submodules, 265
processing, 281, 284
versus managed beans, 321
form fields, representing hidden form fields, 397–398
form handler names, changing for JSF migration, 340
form handlers
adding <html link>s to, 247
creating for DownloadAction class, 245
creating for Login Tiles component, 182–183
declaring, 98, 100
declaring in struts-config.xml, 93
for LookupDispatchAction class, 251
holding information about valid, 377
<form-property> tags
disregarding for dynamic forms, 222
using with LazyValidatorForms, 232
<form> tag, using with validations, 202
<formset> tag, specifying locales with, 217
form tags in HTML tag library, using, 89, 381–382
form validation, explanation of, 53
<form-validation> tag, subtags of, 198
FormBeanConfig instance, using with DefaultDynaFormsLoader, 302
form-beans and action-mappings Sections of struts-config.xml, 181
FormFile class, role in uploading files, 130
forms. *See also* HTML forms
displaying Cancel buttons on, 389–390
preventing from being resubmitted, 450
forms and form handlers, using with Registration webapp, 85–86
forward EL tags, using, 440–441
<forward> in struts-config.xml, calling Tiles definitions from, 168
<forward> tags
attributes of, 98, 100
migrating to JSF, 341
using with <action> tags, 100
ForwardAction class, using, 247–248
forwarding paths, defining with global forwards, 97
Foundations of Ajax website, 6, 358
frames, rendering HTML frames, 396–397
FreeMarker website, 91
full.jsp
adding Find function to, 187, 192
populating and displaying, 158

G

get EL tags, using, 463–464
get() functions, using with DynaActionForms, 227
getAsString EL tags, using, 464, 465
getAttribute() function, relationship to servlet classes, 20
getContainer(), preventing grafting of UI components by, 331
getContextPath() function, using with stylesheets and Tiles layouts, 168
getDateFormat() function, using with MyActionForm, 149
getInputStream() function of FormFile class, role in uploading files, 130

Find it faster at http://superindex.apress.com

getInstance() static function, using in Singleton design pattern, 293
getKeyMethodMap() function,
implementing for LookupDispatchAction class, 251
using with LookupDispatchAction class, 252
getLoader() function
using with DefaultDynaFormsLoader, 302
using in DynaFormsLoaderFactory, 293
getStatusRegular() function, calling on LogonForm, 179
getString() function, using with javax.servlet.jsp.tagext. BodyContent, 28
getters, using with dynamic forms, 225
getters and setters
implementing for ContactForm, 63
using with JSF, 328
getXXX() and setXXX()functions
examples of, 27
using with ActionForm subclass in business logic, 67
global constant, declaring in Validation framework, 204
global exceptions, declaring, 96–97
<global-exceptions> tag, attributes required by, 97
global forwards
calling with include EL tags, 425–426
declaring, 97
using, 258, 405
using redirect EL tags with, 451
greaterEqual EL tags, using, 441–442
greaterThan EL tags, using, 441–442
GUI components. *See* Tiles components

H

handler attribute, using with caught exceptions, 97
handler classes. *See* Java handlers
handlers, using with forms, 85
hasErrors() function, using with Login Tiles component, 186
hash symbol (#), using in JSF syntax, 321
HashMaps, storing (locale,format) pairs in, 148
hashmaps, iterating over, 111
header tag
in Bean tag library, usage of, 90
in EL, 423
Hello World! message, displaying, 28, 29
Hello.jsp, 13–15
helper classes
overview of, 28–29
using in Lisptorq, 362
Hibernate application
downloading, 50, 360
saving beans in, 369
searching objects in, 369–370
using for Registration webapp, 367, 370
using XML description with, 368–369
hidden form fields, representing, 397–398
hidden tag in HTML library, usage of, 89
HQL (Hibernate Query Language), example of, 370
HTML <img>, rendering, 400, 402
HTML buttons, representing, 388–389
html tags
in EL, 398–399
in HTML library, usage of, 89
HTML element, uploading files with, 129
HTML forms, representing, 394, 396. *See also* forms
HTML frames, rendering, 396–397
HTML image input field, representing, 399, 400
HTML links, entering, 404–405
HTML selections, displaying, 411, 417
HTML tag library. *See also* EL tags for HTML tag library
defining prefix for, 83
groups of tags in, 381

in JSF, 320
overview of, 89
purpose of, 381
<html:file> tag, uploading files with, 129
<html:form> tags
setting root object with, 117
using focus=, 86
<html:hidden> tag, localizing validations with, 149
<html:link>s, adding to form handlers, 247
<html:multibox>, creating check boxes with, 236
HTTP requests
encapsulating, 375
generating, 13, 15
HttpServletRequest class
description of, 12
functions associated with, 20
HttpSession class
description of, 12
functions associated with, 20
HttpSession object, obtaining instance of ServletContext from, 244

I

i18n. *See* internationalization
ids of child nodes in JSF, appending prefixes to, 330
if() statement, performing complex validation in JSF with, 329
image EL tags, using, 399, 400
image input fields, showing in JSP, 400
image tag in HTML library, usage of, 89
img tag
in EL, 400, 402
in HTML library, 89
implicit objects, exposing with EL, 121
importAttribute EL tags, using, 466
include EL tags, using, 425–426
include tag in Bean tag library, usage of, 90
IncludeAction class, using, 247–248
indexed fields, using validator with, 207, 209
indexed properties
declaring for dynamic forms, 223–224
description of, 112
using to uploading many files at once, 131
using with EL, 120
validating, 203–204
IndexOutOfBoundsException, throwing, 237
init() function
initializing Struts in, 281
using with DynaFormsPlugIn, 287
using with PlugIn interface, 285
initComponentDefinitions EL tags, using, 467
Initial Bean attribute set, overview of, 385
input attributes
migrating to JSF, 341
using with <action> tag, 99
input pages, calling Tiles definition files from, 168
insert EL tags, using, 459–460
"next" page, displaying for Action subclass, 76
INSTALL directory, creating for Tomcat, 7
instances of classes, controlling with Singleton design pattern, 293
internationalization. *See also* localization
definition of, 143
of legacy applications, 248
relationship to message resource mechanism, 417
relationship to resource mechanism, 414
support in JSF, 324
of webapps, Web resource for, 155
internationalized messages, displaying with message EL tags, 426, 429
Internet Explorer, adding new locale with, 152
intRange,floatRange validator, variables and trigger for, 206
Invoke Application phase in JSF, explanation of, 316
ISO 3166 country codes Web site, 155, 219

Find it faster at http://superindex.apress.com

ISO 8859-1 character encoding, significance of, 151
ISO 639 language codes website, 155, 219
iterate EL tags, using, 442, 444
iterating over hashmaps, 111
iteration tags, using, 109, 111
iterations
  getting information on, 122
  performing with <c:> tag, 122

J

Java classes
  code-centric nature of, 13
  representing events by, 317
Java files, generating for Lisptorq, 364
Java handlers
  implementing for custom validators, 211, 216
  overview of, 26, 28
Java Tag Handler, writing for temperature conversion tag, 33
JavaBeans
  transferring dynamic properties to, 225–226
  using with BeanValidatorForms, 235
JavaScript
  avoiding use of, 250
  generating for client-side validation, 402, 404
javascript tag in HTML library, usage of, 89
JavaServer Pages (JSP), overview of, 13, 15
javax.servlet.http.HttpServletRequest class, overview of, 375
javax.servlet.http.HttpSession class, overview of, 376
javax.servlet.jsp.tagext.BodyContent helper class, description of, 28
javax.servlet.jsp.tagext.BodyTagSupport, subclassing, 26
javax.servlet.jsp.tagext.TagSupport base handler, description of, 26
JNDI (Java Naming and Directory Interface), integrating with Shale, 353–354
JSF (JavaServer Faces)
  backing beans in, 344
  configuring for Registration webapp, 323–324
  creating dynamic Views in, 331
  description of, 107
  displaying error messages in, 330
  events and event listeners in, 315–316, 320
  extending, 320
  focus on View tier, 310
  initializing, 338–339
  internationalization support in, 324
  locating nodes in, 330
  managed beans in, 321
  and method binding, 320–321
  migrating Struts JSP pages to, 339, 341
  overview of, 307–308
  performing complex validation in, 329
  porting Registration webapp to, 322, 336
  relationship to Shale, 308
  request processing lifecycle of, 314, 316
  saving copies of UI components in, 331
  use of server-side UI components by, 311, 314
  using getters and setters with, 328
  using naming containers with, 330
  using treeview UI component with, 330
  versus Shale and Struts, 355
  versus Struts, 308, 310, 313
  and value binding, 320–321
JSF (JavaServer Pages)
  navigating, 321–322
  using Java Strings XE, 321
JSF Central website, 358
JSF download site, 338
JSF equivalents for Struts tags, 340
JSF expression language syntax, relationship to JSTL EL syntax, 321
JSF FAQ website, 358
JSF page and faces requests, 315

JSF pages
creating UI tree for, 312
creating UI trees for, 312
JSF specification website, 336, 357
JSF tag libraries, overview of, 320
JSF taglibs, declaring for Struts-Faces integration library, 339
JSF tags
behavior of, 311
organization of, 311
JSF-type navigation, mixing with Shale's dialog-based navigation, 350
.jsf extension, significance of, 315
JSP global forwards, calling with include EL tags, 425–426
JSP pages and managed beans, one-to-one relationship between, 346
JSPs (Java Server Pages)
declaring Tiles tag library in, 172
writing for temperature conversion tag, 34
effect of variables created on, 20
embedding Tiles components in, 169
request variable in, 15
text-centric nature of, 13
using for layouts in Tiles, 163, 169
using Login tiles component in, 175
using in scopes example, 18
<jsp:forward> tag, example of, 18
<jsp:useBean> tags
example of, 18
variables used with, 20
JSTL (JSP Standard Template Library)
explanation of, 107
resource for, 118
and Struts, 118, 124
tag libraries in, 119
using, 127
JSTL binaries, extracting from Struts distribution zip file, 338
JSTL specs website, 127
JSTL taglibs, declaring for Struts-Faces integration library, 339
JSTL tags, use of #{} in, 342
JUnit page, 358

K

key attribute
using with <bean:message> tag, 84
using with <exception> tag, 97

L

lab sessions
Adding Selected Contacts, 273–274
Collection Listing page, 271
Collection page, 269, 271
Configuring LILLDEP, 102
Contact Entry Page for LILLDEP, 90
ContactForm for LILLDEP, 61, 64
Deleting Selected Contacts in LILLDEP, 236–237
description of, 6
Find facility, 187, 192
implementing ContactAction for LILLDEP, 76
Importing Data into LILLDEP, 140, 142
installing Tomcat, 7, 10
LILLDEP for Malaysian Market, 154
LILLLDEP Full Listing Page, 125, 126
MNC Page, 104
MVC Quiz, 45
Removing Selected Contacts, 272–273
Scopes Quiz, 18–19
Simplifying ContactForm, 126
Struts-Faces Integration Library, 337, 343
Temperature Conversion Tag, 31, 34
Test Driving Lisptorq, 363, 365
Test Driving the DynaForms Plug-in, 302
Up and Down a Search, 275–276
Using JSTL, 127
Using Validator Framework in LILLDEP, 218

Find it faster at http://superindex.apress.com

LabelValueBean class, using with options EL tags, 413–414
language property, using with <formset> tag, 217
languages, scripts in, 144
Latin 1 character encoding, significance of, 151
layout paths versus Tiles definitions, 166
layouts, 161. *See also* Tiles layout definitions
  creating with Tiles, 163, 169
  declaring for Tiles, 162
  defining in Tiles definition files, 167
  defining named attributes in, 461–462
  using stylesheets with, 168–169
LazyValidatorForm JavaDoc, Web resource for, 237
LazyValidatorForms
  and BeanValidatorForms, 235
  disadvantages of, 234
  using, 232, 235
leading slash (/), using with form handlers in JSF migration, 341
lessEqual comparison tags, effect of, 114
lessEqual EL tag, using, 441–442
lessThan EL tags, using, 441–442
LILLDEP (Little Data Entry Program)
  configuring, 102
  contact entry page for, 90
  ContactForm for, 61
  deleting selected contacts in, 236–237
  implementing ContactAction for, 76
  implementing edit facility for, 158
  importing data into, 140, 142
  for Malaysian market, 154
  relationship to MVC design pattern, 45
  using Validator framework in, 218
LILLDEP Declarations for Full and MCN Page Handlers, 261–262
LILLDEP Full Listing Page, 125, 126
LILLDEP main page, using global forwards with, 258
LILLDEP Model classes, autogenerating, 370, 372
LILLDEP webapp, using iterate tags with, 109
link tag in HTML library, usage of, 89
links
  rendering HTML links, 404–405
  switching locales with, 153
Lisptorq
  generating Java files for, 364
  generating LILLDEP Model classes with, 370, 372
  helper classes in, 362
  initializing database for, 365
  running Test program for, 365
  specifying database settings for, 364
  versus Torque, 366
  writing database schema for, 363
  writing test program for, 365
Lisptorq code generator, features of, 360, 362
Lisptorq manual, accessing, 363
Lisptorq software, downloading, 360
Lisptorq tutorial website, 372
Lisptorq user table definition, 361
Lisptorq website, 50
Lisptorq-generated User bean, saving, 362
list-collection.jsp, modifying to select and submit contacts for removal, 273
listeners. *See* events and event listeners in JSF
listing.jsp
  adding check boxes to, 236
  building, 157–158
Listings
  ActionServlet declaration and struts-config.xml file in web.xml, 281
  array lengths set at runtime, 224
  Bean.java code for DefaultDynaFormsLoader, 298, 300
  Beans.java, 297–298
  <c:>...<c:when>, 123
  <c:if> tag, 123

checking for common words as passwords, 210
checking for duplicate user ID, 57
checking for identical fields, 207
class representing mouse clicks, 317
coloring table rows gray with EL and JSTL, 120
coloring table rows to gray, 119
complex validation in RegistrationAction, 73
compose page for uploading files, 136
Counting to 99 scriptlet, 118
Counting to 99 with JSTL, 118
data transformation in RegistrationAction, 74
Database.java modified for Lisptorq and LILLDEP, 372
declaring alphanumeric validator, 215
declaring global exception handlers, 96
declaring global forward, 98
declaring validations for RegistrationForm, 200
DefaultDynaFormsLoader.java, 294, 297
dialogs for Registration webapp (dialog-config.xml), 348–349
Digester used to create multiple users, 290
Digester used to create single user, 288
Display.jsp section, 110
DownloadAction subclassed, 244
DynaActionForm classes, 283
DynaFormsLoaderFactory.java, 291, 293
DynaFormsPlugIn.java, 285, 287
dynamic form bean declaration for entity hierarchy, 278
dynamic form declaration, 222
execute() function, 69
execute() on TilesAction, 170
exists() implementation in Lisptorq, 362
exists() implementation with Hibernate, 370
exists() using Torque, 367
explicit row on indexed field, 208
extends attribute, 279
faces-config.xml for Registration webapp, 323
<fc:forEach> iterating over iterable object, 122
First.jsp, 18
form beans section, 96
form handler declaration, 98
form handler for LookupDispatchAction class, 251
form for using LookupDispatchAction class, 250
form-beans and action-mappings sections of struts-config.xml, 181
GUI widget generating mouse click events, 318–319
Hello.jsp, 13
Hibernate XML description of User bean, 368
hidden format fields (ActionForm), 149
hidden format fields (JSP), 149
index.jsp with Login Tiles embedded component, 176
instance variables prohibited in Action subclass, 68
Java handler class for tag, 27
Java handler to validate alphanumeric strings, 213–214
JSF tags snippet, 312
JSP with taglib declaration and custom tag, 24
LazyVaidatorForm declaring dynamic form, 233
LazyValidatorForm declaring dynamic form, 234
LILLDEP declarations for full and MCN page handlers, 261–262
LILLDEP Model class autogeneration, 371–372
Lisptorq-generated User bean, 362
LocalAction class, 246

Find it faster at http://superindex.apress.com

localizable iterator class, 415–416
localizing validations by brute force, 148
Log4j settings, 260
logging with Apache Commons Logging, 259
logon.jsp, 332, 333
logon.jsp using Shale, 351, 352
LogonForm.java, 176, 178
LookupDispatchAction subclassed, 251–252
<managed-bean> declarations for Registration webapp in Shale, 348
message resource file for Registration webapp in JSF, 324
<message:write>'s handler class, 28
MonkeyPreferences.jsp, 116–117
MonkeyPreferences.jsp Take 2, 117
MouseClickListener interface, 317
MyActionForm snippet, 130
MyDispatchAction, 255
MyLookupDispatchAction with unspecified() and cancelled(), 254
MyLookupDispatchAction with unspecified() and cancelled(), 253
myPage.jsp using simple-layout.jsp, 165
myPage2.jsp with Tiles definition, 166
navigation in RegistrationAction, 75
navigation rule in JSF, 322
new RegistrationAction.java, 231, 232
newuser.jsp view of Login Tiles component, 180
output page for file uploads, 139, 140
plug-in declaration for tiles, 102
plug-in declaration for Validator framework in struts-config.xml, 196
PlugIn interface, 284
PostcodeSearchAction section, 110
PrintingAction, 256
PrintingAction declarations, 256
properties file for Registration webapp, 58
Property.java code for DefaultDynaFormsLoader, 300–301
PropertyUtils used to read ActionForm properties, 242
register.jsp with tags stripped out, 334, 336
Registration webapp skeleton, 359–360
registration.jsp, 82–83, 228
RegistrationAction.java, 71–72
RegistrationForm.java for Validator framework, 199
RegistrationForm.java, 55
regular.jsp link snippet for Login Tiles component, 185
regular.jsp view for Login Tiles component, 178
scriptlets in actions, 15
Second.jsp, 18
SendMessageAction, 137
servlet and servlet mapping declarations, 85–86
Shale's ViewController interface, 344–345
shared Struts configuration file across submodules, 265–266
simple-layout.jsp, 164
simple-layout.jsp with static stylesheet reference, 168
standard signature for custom validation function, 212
Struts configuration files in web.sml, 263
struts configuration files in web.sml, 264
Struts initialized in init(), 281
struts-config.xml file for file uploads, 138
struts-config.xml for new Registration webapp, 229–230
struts-config.xml for Registration webapp, 94–95
submodules declared in web.xml, 264
Subscriber implementing MouseClickListener, 318
success.jsp logoff view for Login Tiles component, 181
taglib section in web.xml, 25

Tiles component with multiple views, 172
Tiles component with single view, 172
Tiles controller with one view, 171
Tiles controller with two views, 171
Tiles declaration in struts-config.xml, 162
Tiles definition file, 163
tiles-def.xml with new definition, 167
TLD file declaring <message:write>, 30
top nodes of Validator XML structure, 197
Torque's XML description of user table, 366
type conversion idiom to read ActionForm properties, 241
uploading fixed number of files, 131
user backing bean in JSF, 325, 328
user backing bean, Take 2, 346–347
User Bean for Digester, 288
User bean for Hibernate, 367–368
User bean saved with Hibernate, 369
user.init() implementation, 354
UserLoginAction.java, 183, 185
Users bean for Digester, 289, 290
validating indexed fields, 208
validation.xml file split into three parts, 197
validwhen validator, 207
varStatus properties, 122–123
web.xml amended for temperature conversion tag, 34
XML file with multiple users, 289
lists of items, creating, 462, 463
load() function, using with DynaFormsLoader, 287
<load-on-startup> value, specifying for Struts-Faces integration library, 338
LocaleActions
switching locales with, 154
using, 245, 247
(locale,format), storing in HashMaps, 148
locale-independent messages, encapsulating, 376
locales
overview of, 146
selecting from browsers, 151
switching with links, 153
localizable iterator class code sample, 415–416
localization. *See also* internationalization
of replacement arguments in Validator framework, 203
of validations, 217, 218
localized images, rendering, 402
localizing
output, 150, 154
validations, 147, 149
Log instances, creating, 259
Log4j
defining priority messages with, 260
significance of, 259
website for, 266
logger instances, creating per class, 259
logging, overview of, 258, 261. *See also* Apache Commons logging
logging functions, managing calls to, 261
<logic::iterate> tag, using, 109, 111
<logic:forward> tag, effect of, 114
<logic:iterate> tag, replacing with <c:forEach> tag, 122
<logic:redirect> tag, effect of, 114
Logic tag library. *See also* EL tags for Logic tag library
attribute sets in, 436, 437
description of, 79
overview of, 109, 115
performing conditional processing with, 113–114
selector attributes in, 437
website for, 127
logical operators, using with EL expressions, 120
logical outcomes in JSF, significance of, 322

Find it faster at http://superindex.apress.com

Login Tile example, 173, 186
Login Tiles component
  displaying views for, 185
  logoff view for, 180
  views for, 178, 181
Login.do and Logoff.do form handlers, using with Login Tiles component, 182
.login definition, declaration for, 182
logoff view, displaying in Login Tiles component, 186
logon.jsp file, rewriting with Shale's Validator framework, 351–352
logon.jsp View, significance of, 332, 334
LogonForms
  calling getStatusRegular() function on, 179
  declaring for Login Tiles component, 182
LogonForm.java code, 176
LookupDispatchActions
  versus DispatchAction class, 254
  role in uploading files, 135–136
  using, 249, 254
LookupDispatchAction subclass, implementing, 251–252

M

Malaysian market, LILLDEP for, 154
managed beans
  in JSF, 321
  and JSP pages, 346
<managed-bean>, relationship to Shale ViewController, 346
<managed-bean> declarations
  including in JSF migration, 343
  using with Dialog Manager in Shale, 349
mapped properties
  declaring for dynamic forms, 224
  using, 112
mapping, relationship to business logic, 76
MappingDispatchAction class, versus DispatchAction subclass, 257–258
mask validator
  purpose of, 202
  variables and trigger for, 206
match conditional processing tag, meaning of, 114
match EL tags, using, 444–445
maxFileSize default setting, description of, 100
maxLength validator, variables and trigger for, 206
message EL tags, using, 426, 429
message resource bundles in JSF
  obtaining, 329
  supporting internationalization with, 324–325
message resource files
  declaring multiple message resource files, 427–428
  user_exists error message in, 329
message resource keys, using with localized labels, 414, 416
message resources section, overview of, 101
message tags
  in Bean tag library, 90
  in HTML library, 381–382
MessageFormat function in Java, performing substitutions with, 330
<message:write> tag
  declaring with TLD file, 29
  example of, 28
messages
  iterator for, 405, 407
  logging, 259–260
messages tag in HTML library, usage of, 89
messagesNotPresent tags, using, 447
messagesPresent tags
  meaning of, 114
  using, 447
<meta> tags, using with HTML pages for character encoding, 147

minLength validator
error message for, 203
variables and trigger for, 206
mnc.jsp, adding Find function to, 187, 193
Model classes. *See also* classes
autogenerating for LILLDEP, 372
creating with Lisptorq, 361–362
Model code, designing first in MVC pattern, 46
ModuleConfig instance, passing to DynaForms plug-in, 283
modules, splitting Struts configuration files into, 264
MonkeyPreferences.jsp, 116–117
mouse click events, generating with GUI widget, 318–319
mouse clicks, representing in class, 317
MouseClickEvent class, alternative to, 319
Mozilla, adding new locale with, 152
multibox tags
in EL, 407–408
in HTML library, usage of, 89
multiple-file-uploading.zip, location and contents of, 140
MVC design pattern
constraints followed by, 39
relationship to Struts, 47–48
separating code by function with, 38
specifying portions of, 46, 47
MyDataForm form bean, using with DownloadAction class, 245
MyDispatchAction subclass code, 255
MyFaces
downloading source for, 336
treeview in, 313
website for, 357
myFunction(), calling with Digester, 290
myPage.jsp, using layout with, 165
myPage2.jsp with Tiles definition, code for, 166
mystartpage.jsp, localizing, 246
myVariable
use of, 20
using StringBuffer for, 20

N

name attribute
using with <action> tag, 99
using with <form-bean> tag, 96
using with <forward> tag, 98, 100
named attributes, defining in layouts, 461–462
namespaces, separating with struts-config.xml, 263
naming containers, using with JSF, 330
native2ascii, role in localizing output, 151
navigation
in business logic for Registration webapp, 70
performing in business logic, 75–76
relationship to business logic, 75
role in business logic, 67
navigation rules, defining in JSF, 322
nest tags, using, 453
nested properties
declaring for dynamic forms, 225
declaring for LazyValidatorForms, 232
description of, 112
using with EL, 120
validating, 203, 204
Nested tag library. *See also* tag libraries
description of, 79
overview of, 115, 117, 451–452
website for, 127
<nested:root> tag, specifiying different root object than default with, 117
nesting levels, writing, 454, 455
new keyword, using in Singleton design pattern, 293
newuser.jsp view of Login Tiles component, 180

Find it faster at http://superindex.apress.com

Nexus framework, relationship to Struts OverDrive, 309
nocache default setting, description of, 100
nodes, locating in JSF, 330
notEmpty tags, using, 114, 437, 439
notEqual EL tags, using, 439, 440
notMatch EL tags, using, 444–445
notPresent EL tags, using, 448
null, returning in JSF, 331

O

objects . *See also* root object
  exposing implicit objects with EL, 121
  searching in Hibernate, 369
Observer design pattern, relationship to events and event listeners in JSF, 316
operators, using with EL expressions, 120
option EL tags, using, 412, 413
option labels, localization of, 414, 417
options tags in HTML library, using
org.apache.struts.action.Action class, overview of, 378
org.apache.struts.action.ActionErrors, overview of, 377
org.apache.struts.action.ActionForm class, overview of, 378
org.apache.struts.action.ActionMapping class, overview of, 377
org.apache.struts.action.ActionMessage class, overview of, 376
org.apache.struts.action.ActionMessages class, overview of, 377
org.apache.struts.action.ExceptionHandler class, overview of, 380
org.apache.struts.tiles.ComponentContext class, overview of, 380
org.apache.struts.tiles.TilesPlugin class, instantiating, 162
org.apache.struts.upload.FormFile interface, overview of, 379

P

page processing, lifecycle of, 80, 82
page redirects, performing, 449, 451
page references in business logic, direct versus indirect types of, 76
Page scope, explanation of, 17
page tags
  in Bean tag library, 90
  in EL, 429–430
parameter attributes
  using with <action> tag for IncludeAction and ForwardAction, 247
  using with <action> tag and DispatchAction class, 254
  using with PrintingAction, 256
parameter tag
  in Bean tag library, 90
  in EL, 423
parse() function, using on org.apache.commons.digester.Digester class, 289
password tag
  in EL, 418–419
  in HTML library, 89
password2 field, validation for, 203
path attribute
  using with <action> tag, 99
  using with <forward> tag, 98, 100
peer classes, handling in Lisptorq, 361
persistence, handling in Lisptorq, 361
pipe symbol (|), using as separator, 140
plain text, significance of, 145
plug-in declarations
  including in struts-config.xml file, 162
  using with Validator framework, 196
PlugIn interface
  implementing, 284, 285
  implementing for DynaForms plug-in, 285, 287
<plug-in> tag, <set-property> tags in, 162

plug-ins
anatomy of, 284–285
declaring, 101
DynaForms plug-in, 280
polling Subscribers, inefficiency of, 316
PostcodeSearchAction code sample, 110
present tags
meaning of, 114
using, 448
primitive types, using with dynamic properties, 225
print() function, using with LookupDispatchAction class, 252
PrintingAction code example, 256
priority messages, logging with Log4j, 260
private access, relationship to Singleton design pattern, 293
Pro Jakarta Struts website, 6
Pro Jakarta Velocity: From Professional to Expert website, 6, 91
Process Validations phase in JSF, explanation of, 316
processing, categories of, 48
properties
accessing dynamic properties, 225
declaring for LazyValidatorForms, 232
declaring indexed properties for dynamic forms, 223–224
declaring mapped properties for dynamic forms, 224
declaring nested properties for dynamic forms, 225
declaring simple properties for dynamic forms, 223
reading in ActionForm subclasses, 241
simple properties, 112
types of, 112
properties files
benefits of, 59
configuring with struts-config.xml, 93
using with ActionErrors, 58
property attribute
role in uploading files, 130
using with <html:submit> buttons for LookupDispatchAction class, 250
using with <plug-in> tag, 102
using with Struts tags, 111
Property class, using with DefaultDynaFormsLoader, 302
property keys, using dotted notation with, 59
Property.java code for DefaultDynaFormsLoader, 300–301
PropertyUtils class
downside of, 242
explanation of, 240
functions of, 240
reading ActionForm properties with, 242
using, 241–242
Publisher role in Observer design pattern, relationship to JSF, 316
Publishers
relating event listeners to, 318
relating events to, 319
put EL tags, using, 461–462
putList EL tags, using, 462–463

Q

queries, cloning for Tiles controller, 190

R

radio button input fields, rendering, 408–409
radio tag in HTML library, usage of, 89
range validator, variables and trigger for, 206
readConfig() function, using with DynaFormsLoaderFactory, 294
redirect, relationship to scopes, 17
redirect EL tags, using, 449, 451
Register() in JSF
interpreting return value of, 331
using, 328
register.jsp View, significance of, 334–335

Find it faster at http://superindex.apress.com

Registration webapp
business logic in, 70, 73
configuring with struts-config.xml, 94–95
declaring and installing HTML and Bean libraries for, 83
dialogs for, 348–349
displaying errors in, 87–88
displaying static text in, 84
faces-config.xml file for, 323
main Views in, 332, 336
making into Tiles component, 173, 186
porting to JSF, 322, 336
properties file for, 58
requirements for, 39, 44
user input for, 56
using data input tags with, 86, 87
using dynamic forms with, 228, 232
using forms and form handlers with, 85–86
using Hibernate for, 367
using Torque for, 366
View component of, 82, 88
registration.jsp code sample, 82–83
RegistrationActions, 71–72
complex validation in, 73
data transformation in, 74
modifying for dynamic form, 231
navigation in, 75
RegistrationForms
instantiating in JSF migration, 342
using validate() function with, 209
validating, 199, 203
RegistrationForm.java code, 55
regular expression, relationship to mask validator, 202
Regular Expressions tutorial website, 218
Render Response phase in JSF
explanation of, 316
significance of, 315
Rendering attribute set, attributes and usage of, 385
replacement arguments, localizing in Validator framework, 203
Request scope, explanation of, 17
request variable in JSP pages, explanation of, 15
required validators
purpose of, 202
variables and trigger for, 206
reset tags
in EL, 409–410
in HTML tag library, 89
reset(), implementing for ContactForm, 64
resolve() function, using with DefaultDynaFormsLoader, 302
resource tags, using, 90, 430–431
ResourceStreamInfo class, using with DownloadAction class, 243
Restore View phase in JSF, explanation of, 315
Review Labs, Editing Contacts in LILLDEP, 157
rewrite EL tag, using, 410–411
rewrite tag in HTML library, usage of, 89
RFC-2822 for email website, 218
root object. *See also* objects
setting with <html:form> tag, 117
specifying, 117
root tags, using, 456–457
runtime exceptions, catching with global exceptions, 96

S

<s:form>s, changing form handler names in, 340–341
<s:loadMessages> versus <f:loadMessages>, 339
save() function, using with LookupDispatchAction class, 252
saveErrors(), signaling complex validation failures with, 74
scope attribute, using with <action> tag, 99

scopes
definition of, 17
relationship to implicit objects, 121
types of, 17
Scopes Quiz lab session, 18–19
scriptlets
advisory about, 108, 112
versus JSTL, 118
Scriptlets in Action listing, 15
Scroller interface, relationship to searching collections, 275
search_results iterator, using, 110
Second.jsp listing, 18
select tags
in EL, 411–412
in HTML tag library, 89
SelectionFormBean, declaring for listing.jsp, 236
selector attributes in Logic tag library, overview of, 437
_self instance, using in Singleton design pattern, 293
server-side UI components, using with JSF, 311
server-side validations, performing in Shale, 351
servlet classes
overview of, 12
starting, 86
servlet containers
basics of, 10–11
behavior of, 13
interaction with custom tags, 24
Servlet specification website, 16
servlet technology
generic nature of, 49
overcoming shortcomings of, 37
significance of, 7
ServletContext
using with DownloadAction class, 244
using with DynaFormsPlugIn, 287
servlets
definition of, 7
website for, 16
session object, storing collection ID on, 273
Session scope
explanation of, 17
storing Collection instance in, 273
sessionScope object, example of, 121
session-scoped variables, storing, 376
set(String propertyName,Object value function
priming form beans with, 224
priming with set(String propertyName,Object value) function, 224
setAttribute() function, relationship to servlet classes, 20
setDateFormat() function, using with MyActionForm, 149
setFile() arguments, role in multiple file uploads at once, 132–133
<set-property> tags
supporting in DynaForms plug-in, 280
using with <plug-in> tags, 162
using with DynaForms plug-in, 303, 305
using with Validator framework, 196
setters and getters, using with JSF, 328
Shale
and Ajax, 354
design objectives of, 309
Dialog Manager in, 348, 350
disadvantage of, 355
features of, 343
integration with Validator framework, 350, 352
JNDI integration with, 353, 354
overview of, 308, 310
relationship to JSF, 308
support for unit testing, 355
versus JSF and Struts, 355, 356

Find it faster at http://superindex.apress.com

ViewController interface in, 344, 348
website for, 357
Shale's dialog-based navigation, mixing with JSF-type navigation, 350
shared declarations, advisory about, 265
simple definition, using with Tiles, 166
simple properties
declaring for dynamic forms, 223
example of, 111
simple Tiles definitions file, header and footer attributes of, 166
simple validations
central class for, 54
explanation of, 53
processing, 53
simple-layout.jsp, 164
relative placement of header, body, and footer in, 164
using with myPage.jsp, 165
with static stylesheet reference, 168
single quotes (‘), escaping in JSF, 325
Singleton design pattern, using with DynaFormsLoaderFactory, 293
size tags
in Bean tag library, 90
in EL, 431–432
source control, enhancing by splitting up struts-config.xml, 263
Spring framework website, 6
Spring website, 358
SQL queries, using with collections, 269
square brackets ([]), declaring indexed properties with, 223
state information, storing for View tier, 311
static text, displaying in Registration webapp, 84
StreamInfo class
implementing with anonymous class, 244
using with DownloadAction class, 243
StringBuffer versus String, explanation of, 20
Struts
benefit of, 7
benefits of, 64
deficiency related to Model portion of MVC pattern, 49
development of, 4
importance of, 310
initializing in init(), 281
relationship to MVC design pattern, 47–48
two-level nested hierarchy of, 311
users of, 5
value binding in, 320
versus JSF, 313
versus JSF (JavaServer Faces), 308, 310
versus JSF and Shale, 355
versus Shale, 308, 310
website for, 305
Struts attribute set, attributes and usage of, 385
struts- config.xml, splitting up, 263, 266
Struts configuration files, sharing across submodules, 265
Struts EL tags. *See* EL tags
Struts internals, configuring in struts-config.xml, 93
Struts JAR files, advisory about sharing of, 84
Struts JSP pages
migrating to JSF, 339, 341
processing of, 82
Struts OverDrive
explanation of, 309
website for, 357
Struts Request, lifecycle of, 48–49
Struts Shale. See Shale
struts tag in Bean tag library, usage of, 90
Struts tags
enabling for EL, 124
JSF equivalents for, 340
libraries associated with, 79
using property attribute with, 111
Struts tags not EL-enabled, 128

Struts Ti (Titanium)
description of, 309
website for, 357
Struts' Validator page website, 218
struts-config.xml file
declarations for using LocalAction class, 245–246
declaring ChangeLocal form handler in, 153
declaring Tiles in, 162
editing for Struts-Faces integration library, 338
examining for Login Tiles component, 181–182
including plug-in declarations in, 162
plug-in declaration for Validator framework in, 196
for Registration webapp., 94–95
structure of, 93–95
using with file uploads, 138–139
Struts-Faces library
description of, 337
downloading, 357
editing web.xml and struts-config.xml files for, 338–339
installing, 337, 338
preparing development environment for, 337
using ActionErrors class with, 342
Struts-Faces tags, Struts equivalents for, 339–340
struts-faces.jar file, extracting, 338
stylesheets, using with layouts, 168
submit button, using LookupDispatchAction class with, 254
submit buttons
role in uploading files, 137–138
role in uploading multiple files, 133
using LookupDispatchAction class with, 249
submit tags
in EL, 418
in HTML tag library, 89
Subscriber role in Observer design pattern, relationship to JSF, 316
subscribers, relating events to, 317–318
Subversion website, 358
success.jsp Login Tiles component code, 181
summary() function, implementing with DispatchAction class, 255
Sun JSF download site, 357

T

tag libraries. *See also* Nested tag library
best practices for, 108
in JSTL, 119
Logic tag library, 109, 114
Nested tag library, 115, 117
taglib declaration, example of, 24
tags
applying relative to objects, 115
attributes of, 24
declaring, 23
features of, 23–24
using with layouts, 164
tags prefix, purpose of, 24
Ted Husted's website, 6
temperature conversion tag, deploying and testing, 34
Temperature Conversion Tag lab session, 31, 34
Test program, running for Lisptorq, 365
text, outputting with <c:out> tag, 121
text tags
in EL, 418–419
in HTML tag library, usage of, 89
textarea tags
in EL, 420
in HTML tag library, 89
Tiles
installing, 162–163
for layouts, 163, 169
overview of, 161–162
preparing for Find facility, 189

Find it faster at http://superindex.apress.com

Tiles <definition>, declaring, 171–172
Tiles action mapping, including for Find facility, 190
Tiles components
  creating, 169
  declaring, 162
  declaring multiple views of, 172
  description of, 161, 169
  examples of, 173, 178, 181, 183, 185–186
  getting external form data related to, 187
  using, 172
Tiles controllers
  creating for components, 170
  declaring, 171
  explanation of, 169
  for Login Tiles component, 182
  writing for Find facility, 189, 190
Tiles definitions
  advantages of, 171
  calling from <forward> in struts-config.xml, 168
  defining layouts in, 167
  example of, 163
  versus layout paths, 166
  splitting into more than one file, 163
  writing for Find facility, 191
Tiles DTD (Document Tag Definition), location of, 163
Tiles JSP, writing for Find facility, 191
Tiles layout definitions, subclassing, 167. *See also* layouts
Tiles tag library
  attributes for, 458
  declaring in JSP, 172
  description of, 79
  overview of, 457
Tiles view, creating, 171
tiles-def.xml file
  declaring layouts and components in, 162
  with single definition, 166
<tiles:getAsString> tag, using with layouts, 165
<tiles:insert> tags
  embedding Tiles components into JSP pages with, 169
  page, template, and component properties of, 166
  using with layouts, 164–165
  using Tiles controllers in, 171–172
time display in Hello.jsp, explanation of, 14
TLD (Tag Library Descriptor) file, significance of, 23
TLD files
  establishing logical path to, 24
  overview of, 29, 31
  physical location of, 25
  writing for temperature conversion tag, 33
Tomcat
  installing, 7– 8
  website for, 16
Torque
  downloading, 360
  using for Registration webapp, 366–367
  versus Lisptorq, 366
  website for, 50
Torque's description of user table, 366
trace() function, using with Log instances, 260
transition tokens, relationship to redirect EL tags, 450–451
translated Application.properties files, processing, 151
treeview in JSF pages, state of, 313
treeview UI component, using with JSF, 330
troubleshooting, Tomcat installation, 10
type attribute
  declaring nested properties for dynamic forms with, 225
  using with <action> tag, 99
  using with <exception> tag, 97
  using with <form-bean> tag, 96
  using with dynamic forms, 222

type conversion
disadvantages of using with ActionForms, 241
disadvantages of using with PropertyUtils class, 241

U

UI components in JSF, saving copies of, 331
UI trees
versus ActionForms, 314
creating for JSF pages, 312–313
Unicode
and UTF-8, 144
versus ASCII, 144
definition of, 144
Web site for, 155
unit testing
support in Shale for, 354
website for, 218
unknown() function, calling for Tiles controller, 190
unspecified() function, using with LookupDispatchAction class, 253–254
Update Model Values phase in JSF, explanation of, 316
uploading files. *See* file uploads
URI (Uniform Resource Identifier), relationship to taglib declarations, 24
URL tags in HTML tag library, descriptions of, 383, 381
url validator, variables and trigger for, 206
URLs
calling with include EL tags, 425–426
resolving and rendering, 410–411
useAttribute EL tags, using, 465
Useful Links. *See also* websites
ACID, 373
"Adding Spice to Struts," 305
Ant website, 358
Apache Ant website, 35
Apache Commons Logging, 266
Apache Commons' BeanUtils project, 237
Apache Struts website, 6
BeanValidatorForm JavaDoc, 237
Commons Validator framework project, 218
Craig McClanahan's blog, 6
CSS Zen Garden, 358
Derby project, 373
Digester framework, 305
DynaActionForm supported types, 237
Eclipse website, 35, 305
EL tutorial from Sun, 128
Foundations of Ajax website, 6
Foundations of Ajax, 358
FreeMarker, 91
Hibernate, 50, 373
internationalization of webapps, 155
ISO 3166 country codes, 219
ISO-639 language codes, 219
JSF Central website, 358
JSF FAQ website, 358
JSF specification, 357
JSTL specs, 127
JUnit page, 358
LazyValidatorForm JavaDoc, 237
Lisptorq, 50, 373
Log4j, 266
Logic tag library, 127
MIME type strings, 266
MyFaces website, 357
Nested tag library, 127
Pro Jakarta Struts website, 6
Pro Jakarta Velocity: From Professional to Expert, 91
Pro JSP 2, 4th Edition, 127
ProJakarta Velocity website, 6
Regular Expressions tutorial, 218
RFC-2822 for email, 218
The ServerSide website, 6

Find it faster at http://superindex.apress.com

servlet information, 16
Servlets specification, 16
Shale website, 357
Spring, 358
Spring's web application framework website, 6
Struts OverDrive website, 357
Struts tags, 128
Struts Ti website, 357
Struts website, 305
Struts' Validator page, 218
Struts-Faces integration library, 357
Subversion website, 358
Sun JSF download site, 357
Ted Husted's website, 6
Tomcat website, 16
Torque, 50, 373
unit testing information, 218
Uuencode entry in Wikipedia, 142
Velocity, 91
WebWork website, 357
user backing bean. *See also* backing beans
code sample, 325, 328
using in JSF version of Registration webapp, 325, 332
User bean for Hibernate code sample, 367–368
User class, persisting versions of, 359–360
user declarations, handling with Digester, 289, 291
user ID, checking for ActionForms, 57
user managed bean, creating for JSF version of Registration webapp, 324
user.init() implementation sample code, 353–354
userid field, validating, 203
UserLoginAction Tiles controller, using with Login Tiles component, 186
UserLoginAction.java code for Login Tiles component, 183, 185
users, logging off in Login Tiles component, 185
Users bean, creating for Digester, 289
users of book, assumptions about, 5
UTF-8 character encoding
relationship to Unicode, 144–145
website for, 155
Uuencode entry in Wikipedia Web address, 142

V

validate attribute, using with <action> tag, 99
validate() functions
body of, 56
creating and maintaining, 216
versus extending Validator framework, 195, 211
implementing, 209
implementing for ContactForm, 63–64
using with ActionForm subclass, 54
validation
of nested and indexed properties, 203–204
performing in JSF, 329
validation function, steps for, 212
validation quiz, 60
validation.xml file, splitting into several files, 197
validations
creating client-side validations, 205
customizing, 209, 216
declaring, 200–201
localizing, 147, 149, 217–218
validations.xml, using with dynamic form for Registration webapp, 231
Validator framework
extending, 210–211
integrating Shale with, 350, 352
migrating legacy code to, 216
overview of, 195–196
using, 198, 205
using constants in, 204–205
using with dynamic forms, 227
using with LazyValidatorForms, 232, 235

Validator plug-in, declaring, 196, 197
ValidatorForm, subclassing for RegistrationForm, 200
validator-rules.xml, declaring validator in, 215
validators
  creating, 211
  declaring in validator-rules.xml, 215
  examples of, 205, 209
validwhen validator
  using, 206–207
  using with indexed fields, 207, 209
  variables and trigger for, 206
value attribute
  using in conditional processing, 113
  using with <plug-in> tag, 102
  using with multibox for check box, 236
value bindings in JSF, examples of, 320–321, 328
var tag, regular expression in, 202
variables
  accessing on scopes, 121
  creating on JSP pages, 20
  exposing with define EL tag, 423, 425
variant property, using with <formset> tag, 217
varStatus attribute, getting information on current iteration with, 122
varStatus properties code sample, 122–123
Velocity website, 91
View code, designing first in MVC pattern, 46
View component, of Registration webapp, 82, 88
View tier,
  relationship to JSF, 310
  storing state information for, 311
ViewController interface in Shale, overview of, 344, 348
Views in Registration webapp, 332, 336

W

WAR files (Web Application aRchives)
  contents of, 10
  web applications bundled as, 10
web applications
  definition of, 3
  re-deploying, 10
web.xml
  ActionServlet declaration and struts-config.xml file in, 281
  amending for temperature conversion tag, 33
  configuring for JSF, 323, 324
  declaring multiple Struts configuration files in, 263
  submodules declared in, 264
web.xml file
  contents of, 25
  declaring data source references in, 353
  declaring environment entries in, 353
  declaring Shale dialogs in, 348
  editing for Struts-Faces integration library, 339
webapp pages, mapping to names in struts-config.xml, 93
webapps, using global forwards with, 258
webapps directory, contents of, 10
WEB-INF directories, using with WAR archive files, 10
websites. *See also* Useful Links
  ACID, 373
  Hibernate software, 360
  Hibernate, 373
  internationalization of webapps, 155
  ISO 3166 country codes, 155
  ISO 639 language codes, 155
  JSF (JavaServer Faces), 338
  Lisptorq manual, 363
  Lisptorq software, 360
  Lisptorq tutorial, 372

Find it faster at http://superindex.apress.com

Lisptorq, 373
Torque software, 360, 373
Unicode, 155
UTF-8 and related encodings, 155
Uuencode entry in Wikipedia, 142
WebWork website, 357
wildcards, using, 261, 262
write tags
in Bean tag library, 90
in EL, 434, 435
writeNesting tags, using, 454–455
Writer instance, using with BodyContent, 28

X

XHTML 1.0–conformant page, rendering, 399
xhtml tags
in EL, 420–421
in HTML tag library, 89
XML
reading with Digester, 288, 291
using with Torque, 366
XML descriptions, using with Hibernate, 368
XML file with multiple users code sample, 289

You Need the Companion eBook

Your purchase of this book entitles you to its companion eBook for only $10.

We believe this Apress title will prove so indispensable that you'll want to carry it with you everywhere, which is why we are offering the companion eBook for $10 to customers who purchase this book now. Convenient and fully searchable, the eBook version of any content-rich, page-heavy Apress book makes a valuable addition to your programming library. You can easily find, copy, and apply code—and then perform examples by quickly toggling between instructions and the application. Even simultaneously tackling a donut, diet soda, and complex code becomes simplified with hands-free eBooks!

Once you purchase this book, getting the $10 companion eBook is simple:

1. Visit **www.apress.com/promo/tendollars/**.
2. Complete a basic registration form to receive a randomly generated question about this title.
3. Answer the question correctly in 60 seconds and you will receive a promotional code to redeem for the $10 eBook.

2560 Ninth Street • Suite 219 • Berkeley, CA 94710

All Apress eBooks subject to copyright protection. No part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher. The purchaser may print the work in full or in part for their own non-commercial use. The purchaser may place the eBook title on any of their personal computers for their own personal reading and reference.

Offer valid through 8/20/06.

FIND IT FAST with the Apress *SuperIndex*™

Quickly Find Out What the Experts Know

Leading by innovation, Apress now offers you its ***SuperIndex***™, a turbocharged companion to the fine index in this book. The Apress ***SuperIndex***™ is a keyword and phrase-enabled search tool that lets you search through the entire Apress library. Powered by dtSearch™, it delivers results instantly.

Instead of paging through a book or a PDF, you can electronically access the topic of your choice from a vast array of Apress titles. The Apress ***SuperIndex***™ is the perfect tool to find critical snippets of code or an obscure reference. The Apress ***SuperIndex***™ enables all users to harness essential information and data from the best minds in technology.

No registration is required, and the Apress ***SuperIndex***™ is free to use.

❶ Thorough and comprehensive searches of over 300 titles

❷ No registration required

❸ Instantaneous results

❹ A single destination to find what you need

❺ Engineered for speed and accuracy

❻ Will spare your time, application, and anxiety level

Search now: ***http://superindex.apress.com***

SuperIndex

Apress®
THE EXPERT'S VOICE™